CONFIGURING HETEROGENEOUS WIRELESS SENSOR NETWORKS UNDER QUALITY-OF-SERVICE CONSTRAINTS

ROBERT JOHAN HUBERT HOES

NATIONAL UNIVERSITY OF SINGAPORE

2010

CONFIGURING HETEROGENEOUS WIRELESS SENSOR NETWORKS UNDER QUALITY-OF-SERVICE CONSTRAINTS

ROBERT JOHAN HUBERT HOES

(MSc, Eindhoven University of Technology)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

NATIONAL UNIVERSITY OF SINGAPORE

2010

Acknowledgements

During the years I did my research for this thesis, a number of people have given me precious time to support me in many ways. Without them, I would have never been able to write this thesis.

I would first of all like to express my gratitude to Prof. Twan Basten. He has been an enormous source of inspiration and motivation during the whole journey of my PhD, and earlier when I did my internship and master's project in 2003 and 2004. I first met him in a course about models for digital systems he was teaching. I enjoyed this course quite a lot, and when the time came to do my internship, I approached Twan to enquire for opportunities. This was probably one of the best decisions I have made to date. Twan is pretty much the ideal supervisor. He gave me a lot of his time for discussions, and a tremendous amount of high-quality feedback on my work. Even while I was far away in Singapore for three years of my PhD, I had discussions with him over Skype and email almost every week. Besides all that, he is a really great person, who does anything he can to make life for his students as comfortable as possible.

My years in Singapore would not have been half as good without Prof. Tham Chen Khong. I am very thankful to him for his support and for letting me be part of his Computer Networks and Distributed Systems lab at NUS. Before I came to Singapore, I barely new anything about networking. Prof. Tham was the one who introduced me to the emerging world of Wireless Sensor Networks, and taught me all the basic and advanced skills I needed.

I would also like to thank Prof. Henk Corporaal. Because of his vast experience, Henk managed to make me see my work from many different angles, which usually led to several new insights. Especially in the beginning of my PhD, the early days in Singapore, he gave me a lot of guidance, and also put me in touch with Prof. Tham. Henk shows a lot of passion to do new things, which is highly inspiring for me and his other students.

Also Marc Geilen played an important role. He is the real guru of Pareto algebra, and always provided me with answers to the complex issues I ran into. Owing to his amazing insight, he always manages to pinpoint mistakes that are very hard to spot, and thereby contributed a lot to

the quality of my work.

My gratitude also goes out to my examiners, Profs Koen Langendoen, Johan Lukkien, Lothar Thiele and Lawrence Wong, who provided me with very useful feedback on the draft of this thesis.

Further, I would like to thank my buddies in the CNDS lab in Singapore. I was lucky to find a bunch of people who enjoyed coffee breaks as much as the Dutch, and who taught me a lot about Asian customs and culture. As most people were working on sensor networks, we had many interesting and useful discussions. I really have to mention Yeow Wai Leong in particular, with whom I worked together on the mobile sink algorithm, which has been the base for Chapter 6 of this thesis.

On the TU/e side, where I returned to for the final year of my PhD, I would like to thank my colleagues in the Electronic Systems group for creating a great atmosphere to work in. Thanks especially to Marja and Rian for all the help with administrative issues, and to Sander Stuijk, who seems to know nearly everything and is always ready to give advice or help out.

Finally, I would really like to thank my parents for always supporting me in whatever way possible. And of course Nidhi, for being there with me since we first met in Singapore in 2003, and for helping me through the difficult moments that are part of doing a PhD!

All of you played an important role in my life during the past years. Thanks and keep in touch!

Rob Hoes

March 2010

Contents

Summary

Wireless sensor networks (WSNs) are useful for a diversity of applications, such as structural monitoring of buildings, farming, assistance in rescue operations, in-home entertainment systems or to monitor people's health. A WSN is a large collection of small sensor devices that provide a detailed view on all sides of the area or object one is interested in.

This thesis deals with the configuration problem of a WSN, starting with a heterogeneous collection of nodes in an area of interest, models of the nodes and their interaction, and task-level requirements in terms of **quality metrics**. Examples of quality metrics are end-to-end latencies, the coverage of the area, or network lifetime. We support multiple quality metrics and optimise these under **constraints**. Targeted is the class of WSNs with a single data sink that use a routing tree for communication. We introduce two models of WSN **tasks** – target tracking and spatial mapping – for the experiments in this thesis.

The configuration process is split in five phases. After an initialisation phase, the **routing tree** is formed. We explore the trade-off between two attributes of a tree – the average path length and the maximum node degree – which affect the quality metrics, but also the complexity of the remaining optimisation trajectory. We introduce new algorithms to efficiently construct a shortest-path spanning tree with a bounded node degree.

The next phase determines the **Pareto-optimal configurations** given the routing tree. A configuration contains settings for the parameters (hardware or software settings) of all nodes in the network, plus the quality metrics they give rise to. The Pareto-optimal configurations, represent the best possible trade-offs between the quality metrics. Given the vastness of the configuration space – exponential in the size of the network – a brute-force is impossible. Still our method efficiently finds, under certain conditions, *all* Pareto points, by **incrementally** searching the configuration space, and discarding potential solutions immediately when they appear to be non-optimal. Experimental results show that the practical complexity of this algorithm is approximately linear in the number of nodes in the network, and thus **scalable** to very large

networks. After computing the Pareto-optimal configurations, one that satisfies the constraints is selected, and the nodes are configured accordingly (the selection and loading phases).

The configuration process can be executed in either a **centralised** or a **distributed** way. Simulations show run times in the order of seconds for the centralised configuration of WSNs of hundreds of TelosB sensor nodes. The distributed algorithms take in the order of minutes for the same networks, but have a lower communication overhead.

We further study meta trade-off between the task's quality and the cost of the configuration process itself. A speed-up of the configuration process can be achieved in exchange for a reduction in the quality. We provide **complexity-control** functionality to fine-tune this trade-off.

The final part of this thesis describes methods to **adapt** the configuration to dynamism **at run time** due to, for example, changing network conditions or a sink that moves around. We use **localised** algorithms to maintain the routing tree and reconfigure the node parameters, and we are able to control the quality/cost trade-off by adjusting the size of the locality in which the reconfiguration takes place.

List of Tables

List of Figures

List of Algorithms

Glossary of Terms

Node	An autonomous device that has at least a processor and a communication interface, and usually also sensors (a sensor node).
Wireless Sensor Network (WSN)	A network of usually a large collection of sensor nodes, which are able to communicate over wireless links.
Sink	A special node in a WSN that is assigned to collect the measurements from the sensor nodes.
Task	The function of a WSN, or the job it is supposed to perform, which is placed under certain performance constraints. Example: a target-tracking task is supposed to find and track target objects in a specified area, and report the target locations back to a central node that displays the information to the user.
Routing Tree	A spanning tree over the network with the sink at its root, used for the communication of data from sensors to the sink.
Node degree	The number of child nodes of a node in the routing tree.
Cluster	A cluster is a sub-set of the nodes involved in the task that forms a sub-tree of the task's routing tree.
Leaf cluster	A cluster with the special property that for each node in the cluster, all its descendants in the WSN's routing tree are also included in the cluster.

Parameter	A tunable property in the system, usually a hard- or software setting. Parameters are the only aspects of the system that we can set directly. Examples: transmission power, duty cycle, sample rate. A controllable parameter is a parameter that the configuration system is able to directly control, as opposed to uncontrollable parameters.
Metric	An measurable quantity that serves as an optimisation target. We may place constraints on metrics, or choose to maximise or minimise them. Quality metrics are those metrics that are ultimately important to the task of the WSN. Examples: detection speed, lifetime, coverage degree. Resource metrics measure resource utilisation, which is important when mapping multiple tasks to the same WSN.
Mapping	A function that yields a vector of metrics for a given vector of parameters. A mapping is a quantitative model of a system/WSN.
Incremental mapping	A mapping from metric a vector to another metric vector, typically as to combine multiple clusters in a compound cluster.
WSN configuration	A vector of parameter values and resulting metric values for a WSN.
Parameter space	A set of all possible distinct vectors of parameters for a given node.
Constraint	A user-specified bound on a metric.
Value function	The main objective function; a function that totally orders all quality metrics.
Pareto-optimal configuration (Pareto point)	A configuration that is not dominated by any other configuration, that is, there is no other configuration that is better in at least one quantity (dimension), while at the same time not worst in any of the other quantities.
Adaptation	Updating the WSN configuration at run time, in response to a change in the situation, e.g. changes in the environment, moving nodes, or amended requirements.

List of Symbols and Notations

Pareto Algebra and Extensions

$\bar{c}$	configuration
$\mathcal{C}$	configuration set
Q	quantity
$\bar{c}_0 \preceq \bar{c}_1$	dominance relation: $\bar{c}_0$ dominates $\bar{c}_1$
$\mathcal{S}$	configuration space
$\mathcal{D}$	constraint set
$\min(\mathcal{C})$	minimisation: returns the set of Pareto-optimal configurations in $\mathcal{C}$
$f(\bar{c})$	mapping function applied to $\bar{c}$ (also defined for configuration sets)
$\mathcal{C}_0 \times \mathcal{C}_1$	the free product of $\mathcal{C}_0$ and $\mathcal{C}_1$
$\mathcal{C} \downarrow k$	abstracts the quantity with index k from $\mathcal{C}$ (also for sets of indices)
$\mathcal{C} \cap \mathcal{D}$	constrains $\mathcal{C}$ to $\mathcal{D}$
$\mathcal{C} \bigtriangledown k$	hides quantity with index k from $\mathcal{C}$ (also for sets of indices)
$\mathcal{C} \bigtriangleup k$	unhides quantity with index k from $\mathcal{C}$ (also for sets of indices)
$\mathcal{C}[k]$	the configuration with index k in $\mathcal{C}$
$\bar{c}[k]$	the value of the quantity with index k in $\bar{c}$
$L(\mathcal{C}_\mathrm{R}, \mathcal{C}_\mathrm{A})$	quality loss of approximated Pareto set $\mathcal{C}_\mathrm{A}$ compared to reference set $\mathcal{C}_\mathrm{R}$
$D(\mathcal{C}_0, \mathcal{C}_1)$	quality difference between two Pareto-minimal configuration sets $\mathcal{C}_0$ and $\mathcal{C}_1$

WSN Configuration

$\mathcal{N}$	the set of nodes in the WSN

$\bar{p}$	vector of controllable-parameter values (parameter vector)
$\bar{u}$	vector of uncontrollable-parameter values
$\mathcal{S}_{\mathrm{P}}$	parameter space
$\mathcal{S}_{\mathrm{Pc}}$	controllable-parameter space
$\mathcal{S}_{\mathrm{Pu}}$	uncontrollable-parameter space
$\mathcal{S}_{\mathrm{M}}$	metric space
$\mathcal{S}_{\mathrm{Mq}}$	quality-metric space
$\mathcal{S}_{\mathrm{Mr}}$	resource-metric space
$\mathcal{D}_{\mathrm{q}}$	quality constraint
$\mathcal{D}_{\mathrm{r}}$	resource constraint
F_{q}	mapping to quality metrics
F_{r}	mapping to resource metrics
$\mathcal{S}_{\mathrm{M}}\|_{\bar{u}}$	sub-set of the metric space for a given $\bar{u}$
$\mathcal{S}_{\mathrm{Pc}}\|_{T}$	sub-set of $\mathcal{S}_{\mathrm{Pc}}$ corresponding to the tree T
val	value function
F_{n}, F_{c}, F_{t}	mappings to node, cluster and task metrics respectively
G_{nc}, G_{cc}	incremental node-to-cluster and cluster-to-cluster mappings
$\mathcal{C}_{\mathrm{prod}}$	product set
I_{P}, I_{Mr}, I_{Mq}	sets of indices to the controllable-parameter, resource-metric, and quality-metric quantities

Routing Tree

$\delta(i)$	degree of node i
$\delta_{\max}$	highest node degree in network
Δ	degree target (degree constraint)
$h(i)$	hop count from node i to the sink
$h_{\max}$	highest hop count (longest path) in network

Adaptation

dev	deviation parameter of the routing-tree reconstruction algorithm

Chapter 1

Introduction

The area of wireless sensor networks (WSNs) and the configuration problem that is covered in this thesis, is introduced in this chapter. The first section provides and overview of wireless sensor networks, some examples of their applications, and the challenges with respect to Quality-of-Service provisioning. The configuration problem and the goals of this work are given in Section 1.2, after which an overview of the contributions of this thesis is presented in Section 1.3. Section 1.4 shows a summary of related work available in the literature, after which an overview of the thesis is given in Section 1.5.

1.1 Motivation

During the past decade, Ambient Intelligence, also known as pervasive computing or ubiquitous computing, has become an important topic in university as well as industrial research. In so-called Ambient Systems, devices in the environment surrounding human beings work together and try to assist people in any possible way. The more traditional electronic systems like servers, laptops and handhelds can all be connected in a network; not only with each other, but also with actuators like displays, speakers or even lighting and heating. Given the ever-decreasing size of integrated circuits, it becomes more and more possible to make electronic devices so small that they can easily be hidden in the environment. These devices are usually wireless and battery operated and therefore easy to put into place.

The current trend is to make these devices not only small, but also cheap so that they can be spread around in large numbers. Such devices typically contain sensors to observe humans or to measure properties of the environment like temperature or humidity. The small devices may

be very simple, but by working together in a wireless network they can still be very powerful: a wireless sensor network. Combining the base network of more conventional devices with wireless sensor networks, the system becomes a true Ambient System: intelligence is embedded in the environment.

Wireless sensor networks have received a great deal of attention over the past years. One of the key differences between wireless sensor networks and conventional computer networks is the fact that sensor nodes are very much constrained in energy. Because of this, low energy consumption is one of the main design goals. Another distinguishing factor of WSNs is the highly cooperative nature of the nodes: a group of sensor nodes can be considered as a single entity with a certain task. Further, similar to ad-hoc networks (but to a lesser extent), sensor networks can be dynamic, because nodes may move and enter the field, or simply run out of energy.

A scenario in which a wireless network of sensors is particularly useful is disaster recovery. Picture a building or a larger area being destroyed by an earthquake or another form of violence. People are trapped inside collapsed buildings and need to be rescued as soon as possible. Because the original communication infrastructure is likely to be partially or fully destroyed, rescue workers have to rely on flexible ad hoc methods of communication. And because many places in the area would be poorly accessible, rescuers could use the help of technology to help them find the victims. Small wireless devices may be spread over the area, from outside or by rescuers inside. These devices, a mix of simple and more powerful ones, act as extra eyes and ears for the rescuers, while at the same time providing an instant wireless communication network. On their handhelds, rescuers receive all relevant available information. Moreover, the victims and rescue workers themselves might wear sensors on or even inside the body, to monitor their health.

It is clear that the network being used in this scenario is very heterogeneous: there are various types of small, low-power sensor nodes, as well as handheld devices. This causes the communication to be very diverse and some data streams (like video) have specific constraints. Sensor nodes that have located a victim need to inform the nearest available rescue workers and send them as much information as possible. This is made difficult by the constant movement of rescuers and the dynamic state of the nodes in between. The goals of a system in such a scenario are about providing information: the information should be reliable and complete and should be delivered in a timely manner. Furthermore, the lifetime of the system as a whole should be as long as possible, without replacing devices. These targets can be formalised into Quality-of-Service

(QoS) performance characteristics. Existing literature on the use of WSNs in disaster recovery is available [8, 51].

A recent example of a real, both wired and wireless, sensor system that is currently being developed and tested in The Netherlands is IJkdijk [62]. A country like The Netherlands, having about 27% of its area and 60% of its population located below sea level, heavily relies on dikes and other water-management systems to protect itself from the water. In recent years, dikes broke a number of times, resulting in the flooding of residential areas. Dike failures mostly occur because dikes are too wet, or due to erosion. A system to detect the onset of such dike failures by sensors inside the dikes, such that maintenance work can be carried out in time, might be cheaper and safer than the alternative of over-dimensioning the dike by adding more clay.

Another interesting project focusing on a real and useful WSN application is COMMON-Sense Net [46]. This project aims to help resource-poor farmers in developing countries to monitor their land and crops, such that the use of irrigation can be made more efficient, and for the prevention of pests and diseases.

Such WSN systems are the main source of inspiration for the research in this thesis, which investigates the challenging question of how to properly configure and maintain a heterogeneous wireless sensor network. The networks we consider may contain a diverse set of sensor nodes, each having various capabilities. Furthermore, our WSNs may be integrated with more powerful wireless devices, such as cameras and handheld computers.

In the early years, work on WSNs was mainly concerned with the design of the sensor nodes themselves. Subsequently, a lot of research went into communication schemes, in-network processing techniques and other higher-level issues [31]. However, it is often assumed that the sensor network is homogeneous and static. Combinations of various types of (sensor) nodes are rarely investigated, let alone the problem of optimally configuring such a heterogeneous network.

When designing and deploying a WSN, a lot of choices need to be made. Römer and Mattern [55] give an overview of the extremely large design space of WSNs, which starts with the types of nodes to be used and the deployment of these nodes. The configuration problem that we cover starts at this point: the nodes are in place and ready to start taking orders. However, they first need to form a network, and figure out exactly how to behave. Each node has software or hardware settings that may be tuned to adjust the node's behaviour.

A typical example of such a parameter of a sensor node is the transmission power of its radio.

Changing this parameter has a number of consequences, such as the communication reliability of the link to a neighbouring node, but also its total power usage and thus the lifetime of its energy supply. Another example is the sample rate of a node's sensor – the number of samples it takes in some period of time. A higher sample rate could imply that the user of the network receives more regular updates about what they are monitoring. At the same time, though, this node, as well as the nodes it depends on to relay data to the user, need to transfer more packets of information, and therefore use more energy. As each node may have several such parameters, the configuration space for a whole network of such nodes is enormous: the total number of possible network configurations grows exponentially with the number of nodes.

Since WSNs are increasingly common and practically useful, people's expectations about them are rising as well. Hence, the topic of Quality-of-Service provisioning, which aims to ensure that explicit performance targets are met, is gaining more and more interest. A heterogeneous network might contain many different types of traffic, each type with its own constraints. Conventional networking has a notion of Quality-of-Service that captures these varying requirements in service types, and has methods to make sure the constraints of all data streams are met. Whether the latter is possible depends on the availability of network resources. And since resources are limited in practical situations, trade-offs have to be found between service quality and resource usage. The concept of Quality-of-Service can be generalised to higher levels of abstraction. We may, for example, consider the user-perceived quality of a video clip that is playing on a display, or even the lifetime of (certain parts of) a system. Though some literature is available, QoS provisioning for wireless sensor networks is still a rather new and unexplored field.

Surveys suggest that there is a need for a middleware layer that negotiates between an application and a network to match QoS demands and the availability of WSN resources [10, 71]. This is challenging, because QoS requirements are often conflicting, and furthermore, adequate ways are needed to predict the behaviour and performance of a possibly heterogeneous network of nodes, under various circumstances. The best possible (optimal) trade-offs between the various relevant QoS demands in a heterogeneous and dynamic WSN should be found. And since the configuration space is so large, it is not feasible to simply try all possible configurations and choose the best.

To efficiently solve the complex multi-objective optimisation problem of configuring a WSN, entirely new methods need to be developed. This thesis introduces such a method, which does not only efficiently find optimal configurations for large WSNs that satisfy multiple QoS constraints, it

is also able to cope with and adapt to changes in the network or its surroundings that are imposed by external factors.

1.2 Problem Statement

As wireless sensor networks typically contain a large number of nodes that can be *configured* individually, the full configuration space of a WSN is vast. The WSNs that we study may contain a mix of various types of nodes. In other words, this thesis deals with *heterogeneous* wireless sensor networks. We currently target the class of WSNs that use a routing tree for communication.

A WSN is deployed to carry out a certain *task* on behalf of the owner of the network, referred to as the *user*; examples of practical WSN tasks are given above. The user has expectations about various aspects of the performance of the network executing the task. Examples of such performance characteristics, called Quality-of-Service (QoS) metrics, or simply *quality metrics*, are the time it takes for measured information to reach the user, the reliability of the network, or the lifetime of the network. The user may place *constraints* on any of these quality metrics. The configuration of the network should be such that the achieved level of quality for each quality metric is at least as good as specified in the constraint for the metric. If there is room for an improvement in quality without violating any of the constraints, the configuration should exploit this opportunity. The process that computes and implements the configuration should be efficient in terms of time, processing power and communication, and scalable to very large networks. Furthermore, if anything changes in the network, its environment, or the demands of the user, the configuration should be adapted to the new situation.

Definition 1.1 (Main Objective). The main goal of this thesis, in one sentence, is to deliver *an efficient and scalable method for the configuration and maintenance of a heterogeneous wireless sensor network, such that performance demands are met.* A more formal definition of the objectives and the limitations of the method is given in Section 3.3.

The ultimate goal we envision is to be able to use a WSN as a platform that can be used to run multiple concurrent tasks under QoS control. While it was not our intention to solve this much broader problem in this thesis, we do hint on ways to extend the current work to support multiple tasks.

1.3 Contributions

The main contribution of this thesis is a complete step-by-step procedure to configure a WSN for a given task as described in the problem statement, and maintain the configuration at run time. We focus on networks that employ a routing tree for communication between the sensors and a (single) data sink. The phases of the configuration process are outlined in Section 3.4. This main contribution is sub-divided into the following parts:

- A framework for hierarchical models of a WSN and a task running on the WSN, and models for spatial mapping and target tracking WSN tasks and nodes within this framework (see Chapter 3).

- Given a WSN with a routing tree in place, a scalable algorithm to find the Pareto-optimal configuration, i.e. the settings for each node that lead to the best possible trade-offs between quality metrics (see Chapter 4). This algorithm is optimised for speed and memory usage, and has a centralised as well as a distributed version. Furthermore, the complexity of the algorithm can be controlled: the cost of the algorithm can be improved in exchange of a reduced quality of the solutions.

- An algorithm to create a routing tree in a given network of randomly deployed nodes, such that the conflicting goals of minimising the average path length (from each node to the root) as well as the maximum node degree (over all nodes) are jointly optimised (see Chapter 5). The balance between these two goals can be controlled by the user. Also this algorithm has both a centralised and a distributed version.

- Methods to maintain a configuration that meets all goals, under changes in the WSN's environment or demands from the user (see Chapter 6). Special attention is given to a scenario in which the sink moves around in the network. The method consists of ways to repair and re-optimise the routing tree if needed, and re-analyse and optimise the settings of the nodes. An important feature of the reconfiguration method is that is can be made to run locally as well as globally: the number of nodes that are affected can be controlled.

1.4 Related Work

This section provides an overview of work that is related to the general goals of this thesis. References to other literature that is associated to specific parts of this work are given in the

respective chapters covering these parts.

1.4.1 WSN Configuration

ASCENT [9] is an early self-configuration scheme for WSNs that autonomously forms a multi-hop topology that provides sensing and communication coverage, and is energy efficient. Furthermore, the topology is adapted to cope with dynamics in the environment.

Another example of WSN configuration is given by Lu et al. [38], who look at WSN configuration in their integrated method for node address allocation, and formation and maintenance of a communication backbone of selected nodes. Their main concern is the overhead of the configuration protocol itself, while they do not optimise the performance of a higher-level application, a goal that is central to our approach.

The need for methods that deal with conflicting performance demands and set up a sensor network properly is recognised by others as well. Pirmez et al. [50], for example, suggest a method for selecting a data-dissemination protocol that best suits a given set of network characteristics and performance demands, based on a fuzzy inference system that uses a knowledge base of system behaviour acquired through simulation. Also Delicato et al. [19] and Wolenetz et al. [67] use such a knowledge base to make a match between demands and network protocols.

A major difference with our work is that these efforts choose a mode of operation that is common for all nodes in the network, while we determine settings for each node individually. Moreover, we are able to deal with arbitrarily heterogeneous networks, in which all nodes and their parameters and parameter ranges may be different. We furthermore explore all optimal trade-offs in the multidimensional design space before ultimately selecting a fitting configuration. This allows for easy reconfiguration when the user's demands change.

1.4.2 Multi-objective Optimisation

The Pareto-optimality criterion, which is used in this thesis to define the optimality of trade-offs between multiple objectives, is a general concept that originally comes from economics. The Pareto points of a system precisely capture all the trade-offs in a multi-dimensional optimisation space. In engineering, it is used, for example, in design-space exploration for embedded systems [45, 63]. The development of Pareto algebra by Geilen et al. [23] (also see Chapter 2) offers a very structured way of analysing the design space.

More traditional ways to find Pareto-optimal solutions include genetic algorithms or related

algorithms like tabu search. SPEA [73], SPEA2 [74] and NSGA-II [18] are well-known examples of genetic algorithms that search for the Pareto frontier of a multi-objective optimisation problem. Genetic algorithms are also applied in WSNs for various configuration tasks [30, 68]. Usually, these approaches are centralised optimisation techniques. The exception being MONSOON [7], which is a distributed scheme that uses agents to carry out application tasks, while the behaviour of these agents is adapted to the situation at hand according to evolutionary principles. Also particle swarm optimisation (PSO), another type of evolutionary algorithm, has been applied to WSNs [59]. However, while PSO can handle multiple parameters, it only optimises one objective (in this case energy usage), or a weighted combination of objectives.

The most important difference between our method and the evolutionary approaches is the fact that we are always able to find the complete set of Pareto-optimal solutions for a given WSN model. Furthermore, since we are using knowledge about the structure of the WSN, we are able to selectively search the configuration space, while evolutionary algorithms ignore any such information and are therefore much slower. Moreover, evolutionary algorithms are randomised and the results are never guaranteed to be complete.

Q-RAM [35] is another framework that uses the Pareto-optimality criterion to find QoS trade-offs. However, it does not use algebraic trade-off computation and it focuses on resource allocation for multiple tasks sharing a single resource, which does not directly apply to WSN configuration. Other work [66] formulates a model for cluster-based target tracking as a two-objective optimisation problem. The paper hints at using Pareto analysis to solve it, but does not give a method to compute the Pareto front.

1.4.3 QoS Support in WSNs

Chen and Varshney [10] give an overview of approaches and challenges related to QoS support in WSNs. There are some network protocols that offer QoS support, often based on delay constraints. The Sequential Assignment Routing (SAR) protocol [61] is one of the first attempts to introduce a notion of QoS to sensor networks. It creates and maintains routing trees from one-hop neighbours of a sink node. SAR optimises a certain additive QoS metric and the energy usage for each path. A sensor node generally has multiple paths to the sink, and chooses one of them based on the QoS requirements and available resources on the paths.

SPEED [26] is another well-known protocol that achieves preliminary (soft) real-time communication in sensor networks. SPEED is a lightweight protocol that attains a certain delivery

rate across the network by utilising feed-back control and geographic forwarding.

Akkaya and Younis [2] present an energy-aware QoS routing protocol, in which they look at end-to-end delays. Sensors are grouped in clusters with a gateway node. The paper focusses on QoS routing within a particular cluster, in which the gateway node determines the routing. Real-time and best-effort traffic may coexist in the network, and a bandwidth ratio is used to separate real-time and best-effort traffic. The routing algorithm tries to determine the optimal bandwidth ratio for the best trade-off between real-time and best-effort traffic.

One example of catering for application-level QoS demands is the work by Perillo and Heinzelman [49]. They attempt to guarantee a minimum data-reliability level while maximising network lifetime, by jointly optimising the sensors' sleep/wake schedules and routing.

The problem of WSN configuration with QoS support fits in the broader domain of middleware for wireless sensor neworks. While the need of such a middleware is recognised [43, 56, 71], it is still a mostly open research problem. Our configuration and maintenance method could be seen as a specific type of WSN middleware.

MiLAN [27] is another middleware framework, which utilises a trade-off between application performance and network cost. It is, however, described in more high-level terms, and it is implicit how to actually achieve this trade-off. Other work on middleware for systems similar to WSNs is available from Baliga and Kumar [5], Chiang et al. [11], and Costa et al. [14, 15].

An important difference between our configuration method and the protocols and algorithms above, is that we can handle any number of QoS metrics, and simultaneously optimise the configuration WSN for all these metrics within given constraints. Furthermore, if there is a configuration possible within the constraints, we are always able to find it.

1.5 Thesis Overview

The thesis commences in Chapter 2 with an introduction to Pareto algebra, a mathematical framework and approach to multi-objective optimisation that is heavily used by the algorithms in this thesis. Subsequently, Chapter 3 gives a detailed overview of our hierarchical modelling framework, which includes models for the nodes and task, and the relation between parameters (node settings) and metrics (optimisation targets), and constraints. Furthermore, this chapter contains two example models that are used in the experiments in this thesis. Finally, a formal definition of the objectives of the configuration process, as well as a breakdown of the process into phases are specified.

Chapter 4 constitutes the core of the configuration method: the description, analysis and experimental evaluation of the QoS optimiser. The chapter includes the basic approach, as well as specific implementation details to improve the speed and memory usage of the algorithm. Also explained is how the algorithm, which is initially defined as a sequential algorithm, can be executed in a distributed way on the nodes of the WSN. Next, we describe how the quality of the configurations that are found by the optimiser can be traded for a cheaper execution of the algorithm, and present preliminary ideas about how the optimiser may be used to work with multiple tasks that are simultaneously mapped to the WSN platform. The chapter closes with an experimental evaluation of the algorithms.

Ways to construct a routing tree are introduced in Chapter 5. The chapter contains centralised and distributed algorithm to construct a routing tree with a given root node on a network of randomly deployed nodes. All aspects of the algorithms are analysed and evaluated by simulation. An overview of results on the full configuration process (comprising all phases) is given at the end of the chapter.

As WSNs are often dynamic, the configuration may need to be adapted at run time, in order to ensure that all nodes remain connected to the sink, and the quality of service is according to the specifications. Chapter 6 describes efficient methods to reconfigure the network to cope with run-time changes. The practically relevant and interesting case of a mobile sink is treated in detail, and simulations illustrate the feasibility of the approach.

Chapter 7 gives an overview of the thesis and provides pointers for future work.

Chapter 2

Pareto Analysis

Pareto optimality is an important criterion for evaluating potential solutions of a multi-objective optimisation problem. Such a problem has multiple conflicting optimisation objectives, and the relative preferences of the various objectives are usually not known. The concept of Pareto optimality was introduced by the Italian economist Vilfredo Pareto in his work on economic efficiency and income distribution [47]. A solution is said to be Pareto optimal (or Pareto efficient) if no *Pareto improvement* can be made, that is, if there is no improvement possible in any of the objectives of the problem without worsening some of the other objectives. In system optimisation, it is generally accepted that only Pareto-optimal solutions – often called *Pareto points* – are worth considering, and all others can be ignored. The Pareto points of a system precisely capture all the trade-offs in a multi-dimensional optimisation space.

A rigorous mathematical foundation for exploiting Pareto optimality was introduced by Geilen et al. [23]. Their *Pareto algebra* provides a framework to work with sets of *configurations*, the potential solutions to a multi-objective optimisation problem. The main motivation was to be able to compute the Pareto solutions to parts of a problem first, and then combining them. In the design-space exploration for a mobile phone, for instance, system components such as the wireless transceiver, memories and processing elements, are analysed separately where possible, and their Pareto-optimal configurations are then put together in order to find the Pareto points for the system as a whole. Such a step-by-step approach is usually more efficient than an approach that analyses solutions for the whole system all at once. Moreover, where conventional methods (e.g. genetic algorithms [73]) normally give an approximation of the Pareto-optimal set, the Pareto-algebra method is exact: the set of solutions found is guaranteed to be complete and the best possible. Our method to configure an WSN is strongly related to this method and Pareto algebra.

This chapter gives a brief introduction of all the concepts and operations of Pareto algebra that are needed in this thesis (Section 2.1). Section 2.2 shows ways to compare multiple sets of Pareto points. This is needed at a number of places in this thesis, for example when comparing heuristics. A complete and efficient implementation of Pareto algebra, which is also used for the experiments in this thesis, is available from `http://www.es.ele.tue.nl/pareto` and has originally been described by Geilen and Basten [22].

2.1 Pareto Algebra

The basics of Pareto algebra are explained in this section. We also introduce some new notation that is useful for the pseudo-code fragments of the algorithms in this thesis.

2.1.1 Configurations and Minimisation

Consider a system with various aspects of interest holding values in a specific range or domain that is determined by the characteristics of the hardware and its environment. Such a domain is called a *quantity*, which is a set Q of values, with a partial order $\preceq_Q$ (if the quantity is clear from the context, we simply write $\preceq$). If $q_1, q_2 \in Q$, then $q_1 \preceq_Q q_2$ means that the value q_1 is considered at least as good as q_2. The ordering of a quantity allows to express a preference of certain values over others. For a quantity Q that is *totally ordered*, any pair of values in the quantity are mutually comparable under $\preceq_Q$. In this thesis, we use quantities for system aspects that we call *parameters* and *metrics*. Parameters are the "inputs" of the system, while metrics are interesting system characteristics that we can measure; for a more precise definition, see Chapter 3. For example, a sensor node may have a quantity $\mathit{Reliability} = \{20, 40, 60, 80\}$ for a reliability metric, with $80 \preceq 60 \preceq 40 \preceq 20$ ($\preceq$ is equal to $\geq$ for greater-is-better).

A configuration space $\mathcal{S}$ is the Cartesian product $Q_1 \times \ldots \times Q_n$ of a finite number of quantities, and a configuration $\bar{c} = (c_1, \ldots, c_n)$ is an element of such a configuration space. The configuration space holds all possible configurations of a system, given a set of quantities. An example of a configuration space for a sensor node is $\mathcal{S} = \mathit{Lifetime} \times \mathit{Reliability}$, with $\mathit{Lifetime} = \{50, 100, 150, 200, 250, 300\}$ and $\mathit{Reliability}$ as above, is shown in Figure 2.1 (all dots of any colour together). We denote the value of quantity Q in a configuration $\bar{c}$ by $\bar{c}(Q)$. Since the space can be very large, it is desirable to select only potentially useful configurations for further analysis, instead of analysing all possibilities. Pareto analysis is able to make such a selection, given the preferences expressed in the ordering of the values of the quantities.

A dominance relation is used to find configurations that are clearly worse than others and do not have to be considered any further. For $\bar{c}_1, \bar{c}_2 \in \mathcal{S}$, configuration $\bar{c}_1$ is said to *dominate* $\bar{c}_2$, denoted by $\bar{c}_1 \preceq_{\mathcal{S}} \bar{c}_2$, if and only if for every quantity Q_k of $\mathcal{S}$, $\bar{c}_1(Q_k) \preceq_{Q_k} \bar{c}_2(Q_k)$. Dominance is a partial order and hence a reflexive relation: every configuration dominates itself. The irreflexive variant, *strict dominance*, is denoted by $\prec$[1]. Configuration $\bar{c}_1$ dominates an other configuration $\bar{c}_2$, when it is better in at least one quantity and not worse in any of the other quantities. For example, given the configuration space of Figure 2.1 and $\preceq = \geq$ for both quantities, then $(100, 80) \preceq (100, 60)$, which means that we do not have to consider the second configuration. However, $(100, 80) \not\preceq (200, 60)$ and also $(200, 60) \not\preceq (100, 80)$, implying none of the two is clearly better.

Definition 2.1 (Pareto-Minimal Set). A set $\mathcal{C}$ of configurations is *Pareto minimal* iff for any $\bar{c}_1, \bar{c}_2 \in \mathcal{C}$, $\bar{c}_1 \not\prec \bar{c}_2$.

We denote the Pareto-minimal subset of an arbitrary configuration set $\mathcal{C}$ by $\min(\mathcal{C})$ and call the process of computing it *minimisation*. For every configuration in $\mathcal{C}$, there is an element of $\min(\mathcal{C})$ that dominates it. The selected configurations are called *Pareto (optimal) configurations* or *Pareto points*. The Pareto-minimal set is unique for finite sets of configurations. Hence, when using a finite configuration set $\mathcal{C}$, we only need to consider the subset $\min(\mathcal{C})$ and we can ignore all the other configurations. We assume in the remainder of this thesis that all configuration sets that we minimise have finite sizes (while quantities and spaces can be infinitely large).

Return to Figure 2.1 for an example. White points in the figure are considered infeasible (they can not be realised in the real system), and all the others are part of a configuration set $\mathcal{C}$. The dominated points in $\mathcal{C}$ are grey, while the Pareto points (the set $\min(\mathcal{C})$) are drawn in black. The Pareto points lie at the border of the shaded are that encloses all configurations in $\mathcal{C}$. This is why the Pareto-minimal set is often referred to as the Pareto frontier.

2.1.2 Derived Quantities

A system often has metrics that depend on other metrics: high-level metrics could be derived from lower-level metrics, while these lower-level metrics themselves may depend on parameters. For example, the lifetime of a network (high-level metric) depends on the lifetimes of the nodes in the network (low-level metric), which in turn depend on parameters like the transmission power

[1] Note that some authors use the term "dominance" in a slightly different way, for example by defining "$\bar{c}_0$ dominates $\bar{c}_1$" as "$\bar{c}_0$ is strictly better than $\bar{c}_1$". This thesis follows the definition by Geilen et al. [23]

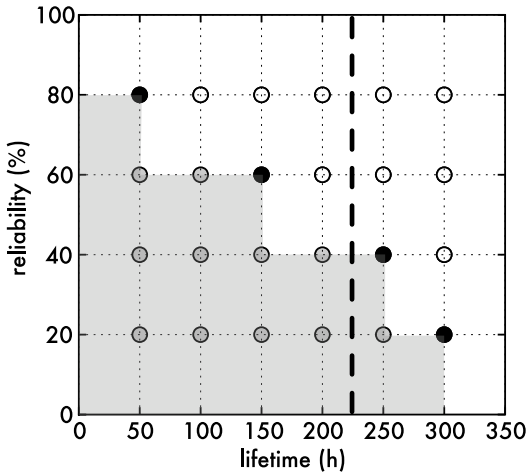


Figure 2.1: An example configuration space for a sensor node, with dominated points (grey), infeasible points (white), and Pareto points (black). The grey and black points together form a configuration set $\mathcal{C}$. The Pareto points in $\min(\mathcal{C})$ dominate all other points in the shaded area. The dashed line represents a safe lower-bound constraint on the lifetime quantity of 225 h. Only the points to the right of the line satisfy the constraint.

levels of the radios in the nodes. For a configuration space $\mathcal{S}$, we define a function $f : \mathcal{S} \to Q$, where the new quantity Q is called a *derived quantity*. In this work, we call f a *mapping function*. We can extend a configuration set $\mathcal{C}$ using f, to create $\mathcal{C}_f = \{\bar{c} \cdot f(\bar{c}) \mid \bar{c} \in \mathcal{C}\}$, where the dot $(\cdot)$ denotes concatenation of tuples. However, an extra restriction needs to be imposed on mapping functions in some cases. Suppose we have two configurations $\bar{c}_1, \bar{c}_2 \in \mathcal{C}$, with $\bar{c}_1 \preceq \bar{c}_2$ and $f(\bar{c}_1) \not\preceq f(\bar{c}_2)$. This would mean that for configurations $\bar{c} \notin \min(\mathcal{C})$, $\bar{c} \cdot f(\bar{c})$ could be in $\min(\mathcal{C}_f)$. This is undesirable, because when minimising before adding the new quantity, potentially optimal configurations may get lost. The key idea of Pareto algebra is that dominated configurations are never interesting and can therefore be removed (by minimising) at any time, at intermediate steps of the analysis. The Pareto algebra approach to optimisation and the method introduced in this thesis depends on this idea.

As a result, mapping functions that are applied after minimisation should be *monotone*.

Definition 2.2 (Monotonicity). Given two partially ordered sets $\mathcal{X}$ with ordering $\preceq_{\mathcal{X}}$ and $\mathcal{Y}$ with ordering $\preceq_{\mathcal{Y}}$, a function $f : \mathcal{X} \to \mathcal{Y}$ is *monotone* iff for any $x_1, x_2 \in \mathcal{X}$, $x_1 \preceq_{\mathcal{X}} x_2$ implies $f(x_1) \preceq_{\mathcal{Y}} f(x_2)$.

This is the generic definition of monotonicity for partial orders. In Pareto algebra, $\mathcal{X}$ would be a configuration space $\mathcal{S}$, and $\mathcal{Y}$ would be a quantity Q or another configuration space. Another term for monotone is *order preserving*, as the definition says that the ordering of the of a partially-

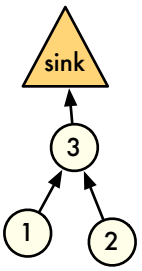


Figure 2.2: A network of three sensor nodes and a sink.

ordered set does not change after the applying the function. A function h on real numbers (with $\preceq$ equal to $\geq$), for instance, is monotone if $x \geq y$ implies $h(x) \geq h(y)$ for all $x, y \in \mathbb{R}$ (h is a non-decreasing function).

For an example of a monotone mapping function, refer to the three-node network in Figure 2.2, where the triangle is the sink that is supposed to receive measurements from the sensors. Each node has a configuration space as in Figure 2.1. Pick a configuration $(\ell_i, r_i) \in \mathit{Lifetime} \times \mathit{Reliability}$ for each node i. We assume for this example that the sink does not need to be configured, as its lifetime would be infinite and reliability is not applicable (the sink does not need to forward the data anymore). A mapping function to compute the lifetime of the network as a whole is $f_\ell(\ell_1, \ell_2, \ell_3) = \min(\ell_1, \ell_2, \ell_3)$, which is monotone. Another high-level metric is the average end-to-end path reliability, which depends on the link reliabilities as follows: $f_i(r_1, r_2, r_3) = \frac{r_3(1+r_1+r_2)}{3}$. Also this is a monotone function. Our WSN models given in Chapter 3 feature both functions.

2.1.3 Other Operations

Free product. A configuration set can be constructed by adding derived quantities, but we can also combine two configuration sets from different spaces. For example, the configuration sets of two sensor nodes may be combined into one joint configuration set. This is done by the *free product* operation. The free product of configuration sets $\mathcal{C}_1 \subseteq \mathcal{S}_1$ and $\mathcal{C}_2 \subseteq \mathcal{S}_2$ is the Cartesian product

$$\mathcal{C}_1 \times \mathcal{C}_2 = \{\bar{c}_1 \cdot \bar{c}_2 \mid \bar{c}_1 \in \mathcal{C}_1, \bar{c} \in \mathcal{C}_2\}, \tag{2.1}$$

which is a subset of the free product of their spaces $\mathcal{S}_1 \times \mathcal{S}_2$. If $\mathcal{C}_1$ and $\mathcal{C}_2$ respectively contain n and m configurations, then $\mathcal{C}_1 \times \mathcal{C}_2$ contains $n \cdot m$ configurations. The free product preserves minimality: $\min(\mathcal{C}_1 \times \mathcal{C}_2) = \min(\mathcal{C}_1) \times \min(\mathcal{C}_2)$.

In this thesis, the free product is used to combine the configuration sets of multiple sensor nodes

into a single configuration set containing all combinations. A configuration in the product set of three nodes with configuration sets as in Figure 2.1, for example, is $(300, 20, 150, 60, 250, 40)$.

Abstraction. After adding derived quantities or combining configuration sets, some quantities in the current configuration set may no longer be necessary. These quantities can be removed by an operation called *abstraction*. If $\bar{a} = (a_1, a_2, \ldots, a_n)$ is a tuple of length n and $1 \leq k \leq n$, then

$$\bar{a} \downarrow k = (a_1, \ldots, a_{k-1}, a_{k+1}, \ldots, a_n). \tag{2.2}$$

Thus, the abstraction operator $\downarrow$ removes one value from the tuple. Likewise, $A \downarrow k = \{\bar{a} \downarrow k \mid \bar{a} \in A\}$. Let $\mathcal{C}$ be a set of configurations of configuration space $\mathcal{S} = Q_1 \times Q_2 \times \ldots \times Q_n$. Then, $\mathcal{C} \downarrow k$ is a set of configurations over configuration space

$$\mathcal{S} \downarrow k = Q_1 \times \ldots \times Q_{k-1} \times Q_{k+1} \times \ldots \times Q_n,$$

so having dimension k removed from each configuration in the set. We also write $\mathcal{S} \downarrow K$, with K a subset of $\{1, 2, \ldots, n\}$, to abstract from multiple quantities at the same time. This is unambiguous, as the order of abstraction is irrelevant. After abstraction, configurations that were previously Pareto optimal may become dominated. Thus, minimisation is required after abstraction in order to ensure that a configuration set is minimal. Consider the Pareto-minimal set $\min(\mathcal{C})$ in Figure 2.1, and abstract away the reliability quantity:

$$\mathcal{C}_{\text{abs}} = \min(\mathcal{C}) \downarrow 2 = \{50, 150, 250, 300\}.$$

The set $\mathcal{C}_{\text{abs}}$ is not minimal; minimising again gives $\min(\mathcal{C}_{\text{abs}}) = \{300\}$.

Constraints. Another important operation of Pareto algebra that is needed to include QoS requirements, is the ability to apply constraints to quantities. A set $\mathcal{D}$ of configurations from configuration space $\mathcal{S}$ is called *safe* if and only if for all $\bar{c}_1, \bar{c}_2 \in \mathcal{S}$ such that $\bar{c}_1 \preceq \bar{c}_2$, $\bar{c}_2 \in \mathcal{D}$ implies that $\bar{c}_1 \in \mathcal{D}$. A safe set of configurations is also called a *safe constraint*. Applying a safe constraint $\mathcal{D}$ to a configuration set $\mathcal{C} \subseteq \mathcal{S}$ yields configuration set $\mathcal{C} \cap \mathcal{D}$. Unsafe constraints go against the fundamental idea that dominated configurations are never to be preferred over Pareto-optimal configurations. Moreover, applying an unsafe constraint after minimisation may result in the loss of Pareto points. For example, given the configuration space $\mathcal{S}$ and set $\mathcal{C}$ in Figure 2.1 (grey

and black points), and a unsafe constraint $\mathcal{D}_{\text{unsafe}} = \{\bar{c} \mid \bar{c}(\mathit{Lifetime}) \leq 225, \bar{c} \in \mathcal{S}\}$ (all points left of the dashed line are included). Then, $\min(\mathcal{C} \cap \mathcal{D}_{\text{unsafe}}) = \{(200, 40), (150, 60), (50, 80)\}$, but $\min(\mathcal{C}) \cap \mathcal{D}_{\text{unsafe}} = \{(150, 60), (50, 80)\}$ so we have lost one point.

Therefore, if we want to minimise intermediate results, only safe constraints should be used. Also, a safe constraint preserves minimality. An example of a safe constraint for a quantity $Q \subseteq \mathbb{R}$ that has a greater-is-better order is a lower-bound constraint, such as $[225, \ldots)$. A safe constraint in Figure 2.1 is $\mathcal{D}_{\text{safe}} = \{\bar{c} \mid \bar{c}(\mathit{Lifetime}) \geq 225, \bar{c} \in \mathcal{S}\}$ (the points to the right of the dashed line). The two Pareto points to the right of the line, $(250, 40)$ and $(300, 20)$, form the Pareto-minimal set of the constraint-satisfying points, $\min(\mathcal{C}) \cap \mathcal{D}_{\text{safe}}$, which is equal to $\min(\mathcal{C} \cap \mathcal{D}_{\text{safe}})$.

2.1.4 Pareto Algebra in Algorithms

Hiding. In algorithms that use Pareto algebra it is often convenient to have some extra information attached to configurations that is not taken into account in operations such as minimisation. This is useful, for example, to separate parameters and metrics in our algorithms in Chapter 4. In these algorithms, metrics are used for computations and dominance checking, while the parameters remain part of the tuple and can therefore easily be found back after a final configuration has been chosen. To facilitate this behaviour, we use an operation called *hiding*: $\mathcal{C} \bigtriangledown k$ hides quantity k from all configurations in configuration set $\mathcal{C}$. It behaves just like abstraction, but the hidden quantities are not actually removed, but remain as meta-information. These quantities are effectively hidden to all operations, and minimisation in particular. Similarly, we can resurrect a quantity by the *unhide* operator: $\mathcal{C} \bigtriangleup k$. The operators are also defined for individual configurations – $\bar{c} \bigtriangledown k$ and $\bar{c} \bigtriangleup k$ – with analogous behaviour. Hiding the lifetime quantity in the configuration set $\mathcal{C}$ of Figure 2.1, and then minimising, results in $\min(\mathcal{C} \bigtriangledown 0) = (50, 80)$.

Now consider the configuration set $\mathcal{C} = \{(1, 1), (2, 1)\}$, and hide the first quantity. If we do not touch the tuples, but simply ignore the first quantity, two quantities remain with the same value in the non-hidden quantity. These configurations dominate each other, while they are not the same, which violates the definition of a partially ordered set. After abstraction of the first quantity, only one configuration remains: (1). To ensure the hide operator properly fits in the theory of Pareto algebra, we therefore keep only one (arbitrary) configuration of the configurations with a common non-hidden part after hiding and remove the others, just like abstraction does (and $|\mathcal{C} \downarrow k| = |\mathcal{C} \bigtriangledown k|$). Note that the implication is that, in general, $(\mathcal{C} \bigtriangledown k) \bigtriangleup k \neq \mathcal{C}$.

Indexing. Another practically useful property is the ability to enumerate configurations sets and select a configuration by its index in the set. In our algorithms, we use square brackets to do this: $\mathcal{C}\,[k]$ returns the k^{th} configuration in the set $\mathcal{C}$. We assume that a configuration set is internally totally ordered (in some arbitrary way) and each configuration in the set is uniquely identified by its index. We use the same notation to index configurations: $\bar{c}[k]$ returns the value of the k^{th} quantity in configuration $\bar{c}$. After hiding a quantity, the indices in the configurations do not change, so a hidden quantity keeps its index (and can be unhidden with it).

2.2 Comparing Pareto Sets

For quality metrics in the WSN models in this thesis, we often use real-valued quantities, which are totally ordered by considering greater values as better. Because of the ordering, it is very easy to compare two values of the same quantity. However, suppose we have a configuration set $\mathcal{C}$ and two approximations of $\min(\mathcal{C})$, and we wish to compare these approximations, and express the difference in a single number. As we are comparing sets of multiple points with trade-offs across various quantities, this is not straightforward. Various performance indices to compare solution sets have been proposed in the literature [44].

We would first like to compare a given approximated Pareto set $\mathcal{C}_{\text{A}}$ for some configuration set $\mathcal{C}$, with the exact Pareto-minimal set $\mathcal{C}_{\text{R}} = \min(\mathcal{C})$ as a reference. This is useful when comparing various heuristic-based methods of approximating the exact Pareto set, used to trade-off analysis speed and accuracy. We employ an adapted version of the *average distance from reference set* performance index [44].

Definition 2.3 (Quality Loss). For a configuration space $\mathcal{S}$ and two Pareto-minimal configuration sets $\mathcal{C}_{\text{R}}, \mathcal{C}_{\text{A}} \subseteq \mathcal{S}$, the quality loss $L(\mathcal{C}_{\text{R}}, \mathcal{C}_{\text{A}})$ of $\mathcal{C}_{\text{A}}$ compared to $\mathcal{C}_{\text{R}}$ is

$$L(\mathcal{C}_{\text{R}}, \mathcal{C}_{\text{A}}) = \frac{1}{|\mathcal{C}_{\text{R}}|} \sum_{\bar{r} \in \mathcal{C}_{\text{R}}} \min_{\bar{a} \in \mathcal{C}_{\text{A}}} d(\bar{r}, \bar{a}). \tag{2.3}$$

The function d returns the normalised distance between two points:

$$d(\bar{r}, \bar{a}) = \frac{1}{k} \sum_{i=0}^{k-1} \frac{[\bar{r}(Q_i) - \bar{a}(Q_i)]^+}{\bar{r}(Q_i)}, \tag{2.4}$$

where k is the number of quantities, and the function $[x]^+$ is zero if $x \leq 0$ and x otherwise. All

quantities contain solely real values with a greater-is-better order.

The distance between two points, $d(\bar{r}, \bar{a})$, is defined as the average relative difference over all dimensions with respect to $\bar{r}$. Dimensions in which $\bar{a}$ dominates $\bar{r}$ are are given a zero relative difference (the closest point to $\bar{r}$ does not need to be dominated by $\bar{r}$, though it will be dominated by at least one point in $\mathcal{C}_\mathrm{R}$, if $\mathcal{C}_\mathrm{R}$ is the exact Pareto set). For each point $\bar{r}$ in the reference set, the closest point $\bar{a}$ in the approximated set is found, and the average distance over the resulting pairs is computed. Negative distances are set to zero, and thus, the index counts only quality loss. The index is a value in the range [0,1], where the value 0 means that set $\mathcal{C}_\mathrm{A}$ contains for any point $\bar{r}$ in the reference set a point $\bar{a}$ that dominates it. That is, $\mathcal{C}_\mathrm{A}$ is at least as good as $\mathcal{C}_\mathrm{R}$ (which typically cannot be expected when approximating $\mathcal{C}_\mathrm{R}$). An index value of q roughly means that on average, for every point $\bar{r}$ in $\mathcal{C}_\mathrm{R}$ the nearest point to $\bar{r}$ in $\mathcal{C}_\mathrm{A}$ has metrics that are a factor q lower than those of $\bar{r}$.

Note that the function L is not symmetric with respect to the configuration sets it compares. If all points in the reference set $\mathcal{C}_\mathrm{R}$ are dominated by points in $\mathcal{C}_\mathrm{A}$, then $L(\mathcal{C}_\mathrm{R}, \mathcal{C}_\mathrm{A}) = 0$ (where typically $L(\mathcal{C}_\mathrm{A}, \mathcal{C}_\mathrm{R}) \neq 0$). If two sets have points that are not dominated by points from the other set, for example when comparing two approximated sets, it is meaningful to look at the difference.

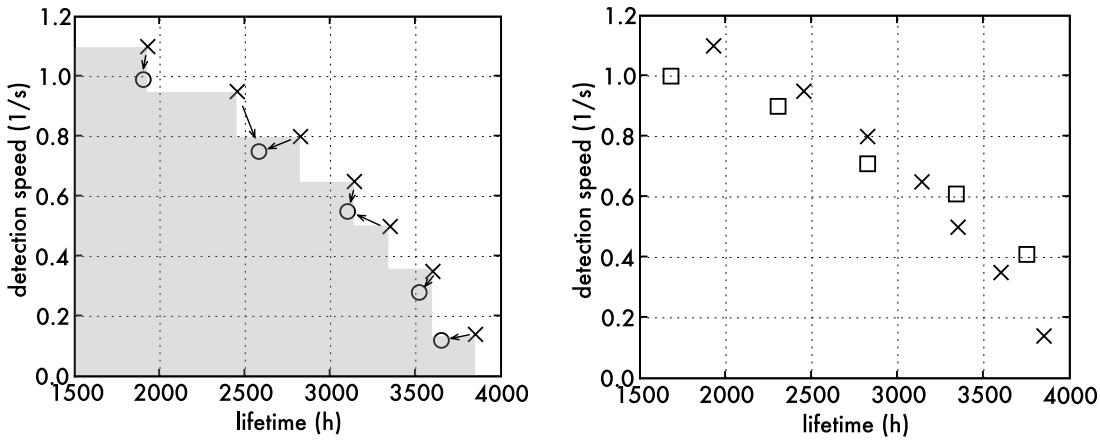
Definition 2.4 (Quality Difference). For a configuration space $\mathcal{S}$ and two Pareto-minimal configuration sets $\mathcal{C}_0, \mathcal{C}_1 \subseteq \mathcal{S}$, the quality difference between the two sets is

$$D(\mathcal{C}_0, \mathcal{C}_1) = L(\mathcal{C}_0, \mathcal{C}_1) - L(\mathcal{C}_1, \mathcal{C}_0). \tag{2.5}$$

If $D(\mathcal{C}_0, \mathcal{C}_1)$ is positive, $\mathcal{C}_0$ may be considered better than $\mathcal{C}_1$, and vice versa.

To be useful, the definition of quality difference must satisfy the minimum requirement for an indicator that compares two Pareto-set approximations: if a configuration set $\mathcal{C}_0$ completely dominates a configuration set $\mathcal{C}_1$, that is each point in $\mathcal{C}_1$ is dominated by a point in $\mathcal{C}_0$, then $D(\mathcal{C}_0, \mathcal{C}_1) \geq 0$ (indicating that $\mathcal{C}_1$ is not better than $\mathcal{C}_0$). See the work of Zitzler et al. [75] for more results on such indicators.

Proposition 2.1 (Requirement for Pareto-set comparison). *If each configuration in a configuration set $\mathcal{C}_1 \subseteq \mathcal{S}$ is dominated by a configuration in another configuration set $\mathcal{C}_0 \subseteq \mathcal{S}$, the quality difference $D(\mathcal{C}_0, \mathcal{C}_1) \geq 0$.*



(a) Quality Loss: $L(\mathcal{C}_\times, \mathcal{C}_\circ) = 0.067$

(b) Quality Difference: $D(\mathcal{C}_\times, \mathcal{C}_\square) = 0.036 - 0.017 = 0.019$

Figure 2.3: Quality Loss and Difference.

Proof. For two configurations $\bar{c}_0, \bar{c}_1 \in \mathcal{S}$, if $\bar{c}_0 \preceq \bar{c}_1$, then by (2.4), $d(\bar{c}_0, \bar{c}_1) \geq 0$ (the normalised distance is never negative), while $d(\bar{c}_1, \bar{c}_0) = 0$ (for each quantity i, $\bar{c}_1(Q_i) \leq \bar{c}_0(Q_i)$, and thus the numerator of (2.4) is zero for all i). Hence, if each $\bar{c}_1 \in \mathcal{C}_1$ is dominated by some configuration in $\mathcal{C}_0$, we are sure that $\min_{\bar{c} \in \mathcal{C}_0} d(\bar{c}_1, \bar{c}) = 0$, and therefore by (2.3), $L(\mathcal{C}_1, \mathcal{C}_0) = 0$, while $L(\mathcal{C}_0, \mathcal{C}_1) \geq 0$. This implies that $D(\mathcal{C}_0, \mathcal{C}_1) = L(\mathcal{C}_0, \mathcal{C}_1) - L(\mathcal{C}_1, \mathcal{C}_0) \geq 0$. □

Figure 2.3 shows examples of the concept of quality loss and difference in a configuration space of two quantities, *Lifetime* and *DetectionSpeed*. In Figure 2.3(a), the configuration set drawn with cross markers is the reference set, while the other one is an approximated set. The arrows indicate which points in the approximated set are nearest to the points in the reference set. These are the distances, determined by (2.4), that are averaged to compute $L(\mathcal{C}_\times, \mathcal{C}_\circ)$ equal to 0.067 in the example. The shaded area represents the part of the configuration space that is dominated by the reference set; it is clear that the approximated set is completely dominated by the reference set, and therefore $L(\mathcal{C}_\circ, \mathcal{C}_\times) = 0$. Figure 2.3(b) shows two Pareto sets that do not dominate each other. As $D(\mathcal{C}_\times, \mathcal{C}_\square)$ is positive, set $\mathcal{C}_\times$ is considered better than set $\mathcal{C}_\square$.

2.3 Summary

This chapter gives a brief introduction to the concept of Pareto optimality and its importance for solving the multi-objective optimisation problem we encounter in the search for suitable WSN configurations. It also contains an overview of Pareto algebra, a mathematical framework and

accompanying optimisation strategies targeted at Pareto optimality, and some extra conventions and notation to ease the use of Pareto algebra in algorithms. The following chapters of this thesis make extensive use of Pareto algebra. Finally, a way to compare different sets of Pareto points with each other is introduced.

Chapter 3

The Configuration Process

Configuring a WSN, what exactly does this involve? This chapter lays the foundation that is needed for the configuration algorithms in this thesis. First, in Section 3.1, the configuration space for a WSN task is defined in general. To explore the configuration space in a sufficiently efficient manner, models are needed. Practical models are given in Section 3.2 for two specific tasks: target tracking and spatial mapping. In Section 3.3, precise optimisation goals are specified, and finally the configuration process is defined in a number of phases in Section 3.4.

3.1 The Configuration Space

This section starts by defining a *task*, which is the entity that is to be optimised by the configuration system. It elaborates on the handles that the optimiser can control and what are the effects of adjusting these.

3.1.1 The Network, Tasks and QoS Requirements

The concept of QoS is used in various domains: in networking, we talk about end-to-end connections that may have QoS requirements, and in the Multiprocessor System-on-Chip domain we have hardware platforms that run independent jobs we could place QoS requirements on. We need a meaningful comparable entity in a WSN: an independent, possibly user-initiated program that "runs" on the WSN and has QoS requirements. We define a *task* in the general sense as the interaction between one or more sensor nodes, actuator nodes, and input nodes, located in a certain (target) area, with the aim to achieve a certain predefined goal. A sensor node is equipped with one or more sensors that can take measurements from the node's environment. An actuator may be a speaker or light source, or a display that shows measured data to the user. The user can

initiate a task at an input node, which could be a simple button or switch, or a more powerful device such as a laptop (which is in fact an actuator as well). This task could be a one-time request for information or action, or a request for periodic measurements or actions. Alternatively, a task could be sensor-initiated, caused by some triggering event.

In this thesis, we specifically look at a type of task comprised of a possibly very large number of sensor nodes and one sink node (an actuator/input node), where the nodes are organised in a tree network with the sink at its root. The communication topology is referred to as the *routing tree*. A node i is a *descendant* of a node j in the routing tree, if j is on the path from i to the root of the tree. Conversely, j is called an *ascendant* of i.

We allow sensor nodes of various types and capabilities in the same network, and it is also possible to include dedicated compute nodes without sensors. For example, a query task may address a group of sensors in a certain area that collect and gather data at a leader node (a *cluster head*). The data may be processed partly by the sensors themselves and the group leader, and then communicated via a multi-hop path to a sink node (operated by the user that needs the information), where it is displayed. In the disaster-recovery scenario, sensor nodes are instructed to detect victims and observe the area around them, and report information back to a rescue worker's handheld sink device. Another example is a so-called *sense & respond* system, in which sensors are observing an area (for instance health monitoring sensors in a person's body), process the measurements and communicate commands, based on the result, to an actuator to take a specific action (e.g. release insulin when a diabetic's sugar level is too high).

QoS requirements can be applied to each of the task's components, but are typically applied to the task as a whole, at *task level*. QoS constraints are usually probabilistic and *soft*; a soft requirement has a given bound, but a certain percentage of violations is accepted. We could for example demand that at least a certain percentage of the target area is covered by sensors, that this area is covered for at least $x\%$ of the time and that the reliability of the measured data is at least $y\%$. Or the communication delay is in $x\%$ of the cases smaller than a given bound; data loss is at most $y\%$. QoS constraints are generally considered soft, because the unpredictable nature of wireless networks makes it practically impossible to give hard guarantees.

We do not make any assumptions on the type of placement (deployment) of the nodes in the field (grid, random, or any other design), other than that it must be possible to form a fully-connected network. We do assume that all nodes have similar communication capabilities and that all links are symmetric (we believe that this holds in many cases, especially when distances

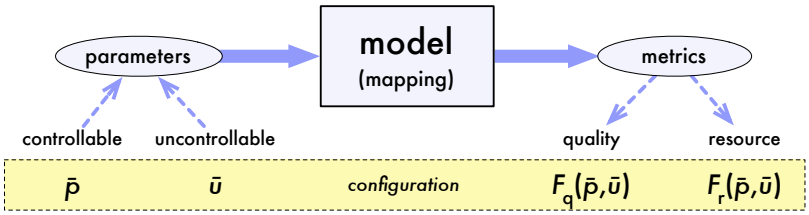


Figure 3.1: Basic structure of a model component. The inputs are parameters*, of which some are* controllable *(a vector $\bar{p}$) and some are not ($\bar{u}$). Measurable behaviour follows from the inputs: the* quality metrics *(performance characteristics that are important to the user) and* resource metrics *(measuring the usage of physical resources).*

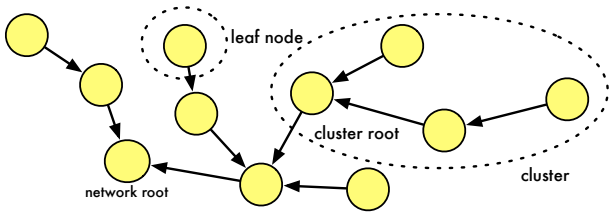


Figure 3.2: A network with an example cluster, as well as root and leaf nodes.

are short and the transmission power sufficiently high); asymmetric links are not yet supported by our configuration method.

3.1.2 Model Components

To analyse a task and its expected behaviour models are needed. We use a hierarchical system of requirements and hardware parameters, where the task that runs on the network forms the highest level (the *task level*) and each node is an entity at the lower *node level*. Intermediate levels are used for groups of nodes called *clusters*.

Definition 3.1 (Cluster). A cluster is a sub-set of the nodes involved in the task that forms a sub-tree of the task's routing tree (also see Figure 3.2).

Note that also individual nodes, as well as the network as a whole, are clusters. As becomes clear in Chapter 4, we use this hierarchy to incrementally compute metrics from lower to higher levels. This is an important feature of our optimisation method.

For each level of the model hierarchy, we define a *model component*. The structure of a model component is the same for each level and given in Figure 3.1. The inputs are *parameters*, of which some are *controllable* (captured in a vector $\bar{p}$) and some are not ($\bar{u}$). Controllable parameters are hardware or software settings that can be set by the configuration system. These are the "knobs" that should be tuned such that the task-level goals are met. Examples are the sample rate and the transmission power of a sensor node. Uncontrollable parameters usually stem from

the environment and may fluctuate at run time. The contention-loss probability of the wireless channel and the transmission delay are possible examples of uncontrollable parameters. The hierarchy implies that the cluster-level parameters comprise all parameters of lower levels, that is, the parameters of all nodes in the cluster.

Each parameter is bound to a certain domain of possible values. We assume this is a discrete domain of a limited number of values, and it is specified as a quantity in Pareto algebra (see Section 2.1). A transmission-power parameter, for example, could have a quantity $TxPower = \{0, -5, -10\}$, where the values are power levels in dBm. Consequently, all possible vectors of parameters are elements of a parameter space $\mathcal{S}_{\mathrm{P}}$, which is the free product of the parameter quantities. The parameter space is defined as being unordered, since we do not have a preference for any parameter value as such; what matters is the effect of parameters on metrics. We also define separate parameter spaces for controllable and uncontrollable parameters, respectively called $\mathcal{S}_{\mathrm{Pc}}$ and $\mathcal{S}_{\mathrm{Pu}}$, such that $\mathcal{S}_{\mathrm{Pc}} \times \mathcal{S}_{\mathrm{Pu}} = \mathcal{S}_{\mathrm{P}}$.

A certain combination of $\bar{p}$ and $\bar{u}$ vectors leads to measurable behaviour expressed in metrics (the outputs in Figure 3.1). Some of these metrics, the *quality metrics*, are the performance characteristics that are important to the user, such as a delay or a lifetime estimation. These quality metrics may have QoS constraints attached to them, and they are the optimisation targets for the configuration system. An example is the lifetime of a node or the network, or the reliability of the link between a child and parent node (see the next section for more examples). The *resource metrics* reflect the usage of physical resources. For each resource of interest there is a resource metric as well as a resource constraint that specifies a bound on the use of this resource[1]. A resource metrics is, for example, the packet transmission rate of a node. Resource metrics play an important role if multiple tasks, which share resources, are to be mapped to the same network. Each model component, at any level of the model hierarchy, has its own metrics. A node-level model, for instance, could have a quality metric that indicates the reliability of sending a packet to the next node, while a task-level model may include an end-to-end reliability metric. Eventually, the task-level metrics are the only ones that matter, since they give the performance for the task as a whole.

We assume that each metric can be derived from parameters by a *mapping function*, which is defined as a function $f : \mathcal{S}_{\mathrm{P}} \rightarrow Q_f$ that maps a parameter vector from the parameter space $\mathcal{S}_{\mathrm{P}}$ to

[1]Resource constraints are set when the hardware is designed, and can therefore also be seen as design-time parameters. Since we focus on the configuration of pre-selected existing hardware, however, we consider them as fixed resource constraints.

a derived quantity Q_f (see Section 2.1.2 and Section 3.2 for examples). The quantity Q_f contains all possible metric values for any combination of controllable and uncontrollable parameter values, and is partially ordered. The free product of all metric quantities is called the metric space $\mathcal{S}_{\mathrm{M}}$. There are mapping functions for quality as well as resource metrics; the corresponding spaces are denoted $\mathcal{S}_{\mathrm{Mq}}$ and $\mathcal{S}_{\mathrm{Mr}}$ respectively, and $\mathcal{S}_{\mathrm{Mq}} \times \mathcal{S}_{\mathrm{Mr}} = \mathcal{S}_{\mathrm{M}}$. The constraints on quality and resource metrics are defined as feasible and safe (as defined in Section 2.1.3) sub-sets of the metrics spaces: $\mathcal{D}_{\mathrm{q}} \subseteq \mathcal{S}_{\mathrm{Mq}}$ and $\mathcal{D}_{\mathrm{r}} \subseteq \mathcal{S}_{\mathrm{Mr}}$.

Definition 3.2 (Mapping). A *mapping* $F : \mathcal{S}_{\mathrm{P}} \to \mathcal{S}_{\mathrm{M}}$, for parameter space $\mathcal{S}_{\mathrm{P}}$ and metric space $\mathcal{S}_{\mathrm{M}}$, derives a vector of metrics from a vector of parameters. More precisely, F is a tuple of mapping functions $f_i : \mathcal{S}_{\mathrm{P}} \to Q_i$, one for each metric i: $F = (f_0, f_1, \ldots, f_{k-1})$, with k the number of metrics. The function F can be lifted to sets: $F(\mathcal{C}) = \{F(\bar{c}) \mid \bar{c} \in \mathcal{C}\}$, with $\mathcal{C} \subseteq \mathcal{S}_{\mathrm{P}}$.

We further denote separate mappings for quality and resource metrics by F_{q} and F_{r} respectively.

3.1.3 Configurations

Our WSN task consists of a large number of nodes organised in a tree network. In general, the metrics defined for such a task (and thus the mappings) depend on the node locations and the way nodes are connected in the tree. We assume that the placement of nodes is beyond our control, and therefore the node locations fit in the model as uncontrollable parameters. The routing tree, on the other hand, while it is obviously restricted by node placements and transceiver capabilities, can be constructed by ourselves (more about this in Chapter 5), and hence we can include a controllable *parent node* parameter for each node. Besides these two parameters, models may vary widely.

Definition 3.3 (WSN Configuration). A configuration for a given model component (parameter spaces $\mathcal{S}_{\mathrm{Pc}}$ and $\mathcal{S}_{\mathrm{Pu}}$, and quality and resource mappings F_{q} and F_{r}) is a tuple

$$\left(\bar{p} \cdot \bar{u} \cdot F_{\mathrm{q}}(\bar{p} \cdot \bar{u}) \cdot F_{\mathrm{r}}(\bar{p} \cdot \bar{u})\right),$$

with $\bar{p} \in \mathcal{S}_{\mathrm{Pc}}$ and $\bar{u} \in \mathcal{S}_{\mathrm{Pu}}$.

Note that a WSN configuration is a configuration as defined in Pareto algebra (see Chapter 2).

At any point in time, the WSN is in a certain configuration (also see Figure 3.1). For a given model, the configuration system can only set the controllable-parameter vector $\bar{p}$, as the vector

of uncontrollables $\bar{u}$ is imposed by external sources. It is therefore useful to consider the subsets of a metric quantity Q_f or space $\mathcal{S}_{\mathrm{M}}$, given the current value of $\bar{u}$. We denote these subsets by $Q_f|_{\bar{u}}$ and $\mathcal{S}_{\mathrm{M}}|_{\bar{u}}$ respectively. Thus, quantity $Q_f|_{\bar{u}} \subseteq Q_f$ (space $\mathcal{S}_{\mathrm{M}}|_{\bar{u}} \subseteq \mathcal{S}_{\mathrm{M}}$) is the set of metric values that results from mapping all controllable-parameter vectors in $\mathcal{S}_{\mathrm{Pc}}$, for a certain uncontrollable-parameter vector $\bar{u} \in \mathcal{S}_{\mathrm{Pu}}$, and this quantity (space) has a partial ordering that reflects the relative preference of configurations in this situation.

Moreover, $\bar{u}$ is subject to changes over time[2], and therefore also the metrics in the current configuration are. Hence, when the controllable parameters of a configuration are kept constant, the metrics of this configuration may move in the metric space. This move could be such, that another configuration would become better than the current one. This suggests that the configuration system should be dynamic and adapt the chosen $\bar{p}$ to the new situation that arises if $\bar{u}$ changes, to ensure that the configuration remains the best possible. Chapter 6 goes into this in detail.

The configuration system needs to select one task configuration from the total configuration space given a vector $\bar{u}$, which means it needs to choose a vector $\bar{p}$ from the space $\mathcal{S}_{\mathrm{Pc}}$ for the task. Hence, the size of the configuration space is equal to $|\mathcal{S}_{\mathrm{Pc}}|$. Note that $\mathcal{S}_{\mathrm{Pc}}$ is the free product of the controllable-parameter spaces of all nodes involved in the task. Suppose each node in a network of n nodes has a controllable-parameter space of size k; the total configuration space for the task then has size k^n, which implies that the complexity of finding a suitable configuration increases exponentially with the number of nodes. Solving this problem efficiently is a central goal of this thesis.

3.2 Spatial-Mapping and Target-Tracking Tasks

Now the basic structure of a task model has been laid out, we introduce two practically useful examples of WSN tasks, which are used in experiments in this thesis. Furthermore, these are elementary sensor-network tasks, which can be used as building blocks for more complex tasks and models. The example models only contain quality metrics; resource models are left as future work.

Consider a network that consists of a collection $\mathcal{N}$ of identical sensor nodes. The nodes are randomly scattered in an area, and do not move once deployed. We define the following two tasks

[2]We could actually write $\bar{u}(t)$, to make the time dependency explicit. In the Pareto analysis, however, we may consider all possible values of $\bar{u}$ and their effects, together, disregarding the time factor. In these cases, we do not use the time index.

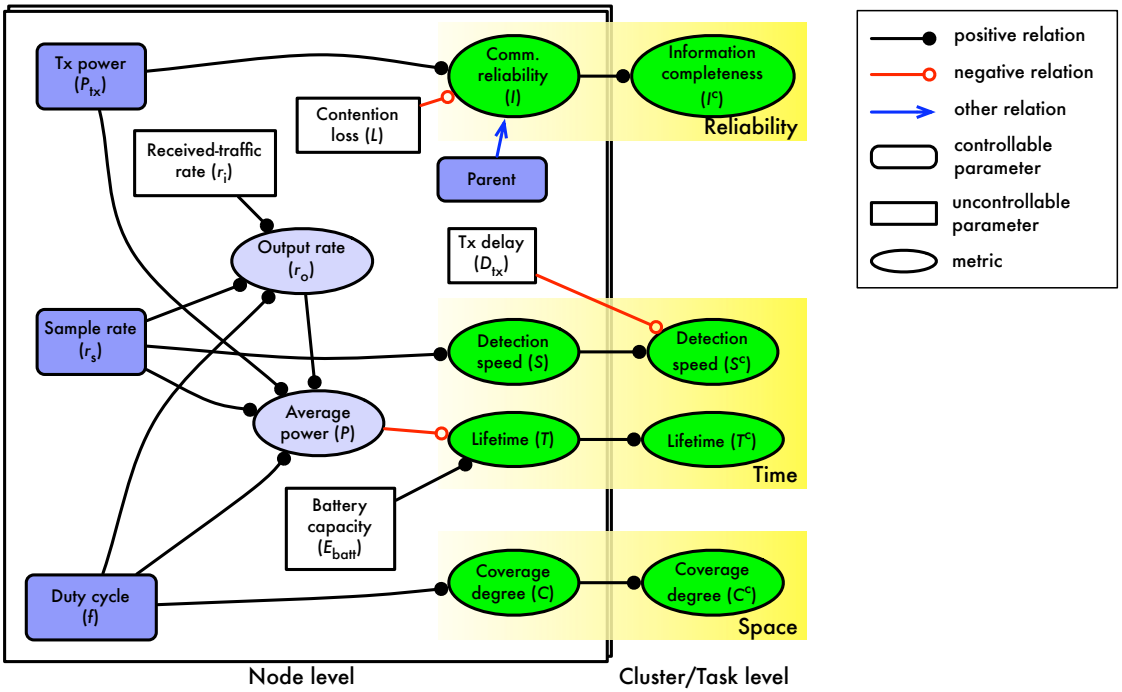


Figure 3.3: Hierarchical trade-off model: relations between parameters (left), node-level quality metrics and task-level quality metrics (in the shaded boxes).

for this network. Firstly, *spatial mapping* (SM), in which all nodes periodically take samples that are sent to the user, for instance to determine the temperature profile over the area. The second task is *target tracking* (TT), in which the objective is to detect and follow target objects. The main difference is that nodes in SM continuously transmit data, while a node in TT only sends a report if it detects a target.

3.2.1 Node-Level Trade-off Models

The left side of Figure 3.3 gives an overview of parameters and how they relate to node-level quality metrics (on the right, in the box). The figure only shows the TT task; as explained below, the parameters of the SM task are the same, while the metrics are slightly different. Rectangles and rounded rectangles represent uncontrollable and controllable parameters respectively, while ovals are quality metrics. Important quality metrics are grouped in three different dimensions: *reliability*, *time* and *space*. Lines with a filled circle at the end represent positive relationships: if the incoming parameter/metric becomes larger, the other also becomes larger. Likewise, lines with an open circle are negative relationships, while lines with an arrowhead are relations that are not clearly positive or negative. The diagram shows that configuring a node means making trade-offs: adjusting parameters has a positive influence on some quality metrics and a negative influence on

Table 3.1: Node-level mappings (F_{n}) for a node n

Reliability

Comm. reliability
$$I(n) = \left(1 - Q\left(\sqrt{\frac{2P_{\mathrm{rx}}(P_{\mathrm{tx}}(n), d)}{N}}\right)\right)^{b}(1 - L) \tag{3.1a}$$
(functions P_{rx} and Q as in (3.2) and (3.3))

Time

Lifetime
$$T(n) = \frac{E_{\mathrm{batt}}}{P(n)} \tag{3.1b}$$

Reporting rate (SM)
$$r(n) = r_{\mathrm{s}}(n) \tag{3.1c}$$

Detection speed (TT)
$$S(n) = \frac{1}{r_{\mathrm{s}}(n)^{-1} + D_{\mathrm{s}}} \tag{3.1d}$$

Space

Coverage degree
$$C(n) = f(n) \tag{3.1e}$$

Additional metrics

Output rate (SM)
$$r_{\mathrm{o}}(n) = r_{\mathrm{i}}(n) + f(n)r_{\mathrm{s}}(n) \tag{3.1f}$$

Output rate (TT)
$$r_{\mathrm{o}}(n) = r_{\mathrm{i}}(n) + m\frac{\pi R_{\mathrm{s}}^2}{A}f(n)r_{\mathrm{s}}(n) \tag{3.1g}$$

Average power
$$P(n) = E_{\mathrm{s}}r_{\mathrm{s}}(n)f(n) + P_{\mathrm{mcu}}f(n) + E_{\mathrm{tx}}(P_{\mathrm{tx}}(n))r_{\mathrm{o}}(n) + E_{\mathrm{rx}}r_{\mathrm{i}}(n) \tag{3.1h}$$

others.

The node-level models for the SM and TT tasks are explained below. For every metric we give a mapping function in Table 3.1, which explicitly defines its relation to the parameters. The relations do not necessarily need to be defined analytically. Other ways to obtain mappings for a set of parameter vectors are, for example, simulations or neural networks. Here, we use explicit equations mainly for speed and ease of use. Our simulations described in Section 3.2.3 show that the mapping functions are sufficiently precise to accurately predict the quality metrics for both tasks.

The first parameter of a node n is its sensor's *sample rate* $r_{\mathrm{s}}(n)$, the number of times per second measurements are taken and processed. The *transmit power* $P_{\mathrm{tx}}(n)$ is the power level at which the node's radio transmits data. Lastly, a node employs a periodic sleep/wake schedule, with fixed period and *duty cycle* $f(n)$, the latter being the fraction of the time the node is awake (active). When the node is in sleep mode, it does not take samples and its micro-controller unit (MCU) is in low-power mode. We assume the transceiver (including the MAC protocol) does its own power management. Finally, each node has a *parent*, which is the node it sends its reports to according to the routing tree. The metrics are defined below, and summarised in Table 3.1.

Reliability. When a packet is transmitted from a child to a parent node, it is received without error with a probability depending on the received signal-to-noise ratio and the size of the packet. According to a path-loss model, the signal power at the receiver depends on $P_{\mathrm{tx}}(n)$ (at the sender) and the distance between sender and receiver d, according to

$$P_{\mathrm{rx}}(P_{\mathrm{tx}}(n), d) = k_r P_{\mathrm{tx}}(n) \left(\frac{d_0}{d}\right)^{\alpha}, \tag{3.2}$$

for constant gain factor k_r, path-loss coefficient $\alpha \geq 2$ and reference distance d_0. Moreover, transmissions may interfere with transmissions of other nodes. For simplicity, we assume a constant contention-loss probability L. Then, the *communication reliability* $I(n)$ (the probability of correctly transferring a data packet) assuming a fixed packet size b (in bits) and QPSK signalling, is expressed as (3.1a) in Table 3.1, with

$$Q(x) = \frac{1}{2}\mathrm{erfc}(\frac{x}{\sqrt{2}}), \tag{3.3}$$

and noise level N [24]. Hence, we need to know the distance from sender to receiver, which could be obtained from knowledge of the deployment, or measurement by the node itself (e.g. [53] or [25]). A more practical way may be to simply have an calibration phase for parent and child, in which for various transmission-power levels, the received power, or even the communication reliability, is directly measured. In the latter case, the mapping function for $I(n)$ simply becomes a look-up table.

Time. The node's battery has a limited capacity E_{batt}. The *lifetime* $T(n)$ of a node follows from the average power level $P(n)$ via (3.1b). We define the average power $P(n)$ as (3.1h), with constant energy to take and process a sample (E_{s}), power of the MCU in active mode (P_{mcu}), and energy to transmit and receive a packet (E_{tx} and E_{rx}). The energy to transmit a packet depends, among other things, on the transmit power, which is why it is written as a function of $P_{\text{tx}}(n)$ in (3.1h). For the precise relation, one generally needs to revert to the datasheet of the transceiver; see Table 3.4 for a conversion table for TelosB sensor nodes. We assume the transceiver uses a MAC protocol that minimises idle listening, such as B-MAC [52]. The power level depends on all three parameters, as well as on the additional metric *output traffic rate* $r_{\text{o}}(n)$, which is the average rate of packet transmissions ((3.1f) for SM, (3.1g) for TT). This rate includes the node's own generated traffic, but also traffic that is to be relayed on behalf of other nodes: the *received traffic rate* $r_{\text{i}}(n)$. Since TT nodes only transmit when a target is in range, $r_{\text{o}}(n)$ does not only depend on the sample rate and duty cycle, but also on the target's trajectory. The fraction of the time that a target is expected to be near is assumed to be equal to the number of targets m times the fraction of the total area A that is covered by the sensor (with sensing range R_{s}).

A quality metric unique to the SM task is the *reporting rate* $r(n)$, a measure of the rate at which the spatial map is updated. It is taken equal to the sample rate in (3.1c). Since TT nodes do not continuously report information, the reporting-rate metric is not used. Instead, we are interested in the *detection speed*. We define the detection delay for a sensor node in the active mode, as the time it takes from the arrival of a nearby target until its detection. This delay depends on sample rate $r_{\text{s}}(n)$, and (combined in the constant D_{s}) the duration of sampling and detection. The (worst-case) detection speed S, given by (3.1d), is the inverse of the detection delay. The detection delay and reporting rate are defined for the node in the active mode; the trade-off between active and sleep is expressed in the coverage-degree metric below.

Table 3.2: Cluster-level mappings (G_{nc}) for a cluster c

Reliability			
	Information completeness	$I^{\mathrm{c}}(c) = \dfrac{\sum_{i \in c} \prod_{j \text{ on } \vec{p}^i} I(j)}{\lvert c \rvert}$	(3.4a)
Time			
	Reporting rate (SM)	$r^{\mathrm{c}}(c) = \dfrac{\sum_{i \in c} r(i)}{\lvert c \rvert}$	(3.4b)
	Detection speed (TT)	$S^{\mathrm{c}}(c) = \min\limits_{i \in c} \dfrac{1}{S(i)^{-1} + \lvert \vec{p}^i \rvert D_{\mathrm{tx}}}$	(3.4c)
	Lifetime	$T^{\mathrm{c}}(c) = \min\limits_{i \in c} T(i)$	(3.4d)
Space			
	Coverage degree	$C^{\mathrm{c}}(c) = \min\limits_{i \in c} C(i)$	(3.4e)

Space. A sensor is said to *cover* the area within its sensing range when it is active. Since a sensor node is typically asleep most of the time, it does not continuously cover this area. We therefore define the metric *coverage degree* $C(n)$ in (3.1e) as the fraction of the time the sensor is switched on.

3.2.2 Task- and Cluster-Level Trade-off Models

The mapping functions for the task- and cluster-level metrics, as shown in Table 3.2, are explained below. The functions in the table are for a cluster c, and use the node-level metrics of Table 3.1 as a short-cut, instead of deriving them directly from the parameters (which is of course possible as well). Task-level mapping functions are obtained by substituting the network $\mathcal{N}$ for c.

Reliability. In both scenarios, nodes send reports to the user. However, because the communication of reports usually has a limited reliability, not all reports may reach the destination. We want to know how complete the data is that is received by the user: the *information completeness* $I^{\mathrm{c}}(c)$ is the fraction of all generated reports that arrive. This is approximately equal to the average end-to-end communication reliability over all nodes, given by (3.4a), where $\vec{p}^i$ is the path from node i to the root node.

Time. For an SM task, we are interested in the *reporting rate* $r^{\mathrm{c}}(c)$ of the network/cluster, which is defined by (3.4b) by the average reporting rate over all nodes. A common timeliness metric of a WSN that does target detection is the time it takes from the appearance of a target until the detection report reaches the user. For each node, this delay depends on its detection speed and the hop count $|\vec{p}^{\,i}|$ to the root node. The worst-case *detection speed* $S^{\mathrm{c}}(n)$ is given by (3.4c), where D_{tx} is the transmission delay (including MAC delay).

Further, we use (3.4d) for the *lifetime* $T^{\mathrm{c}}(c)$, a definition that considers all nodes in the network/cluster as essential. This definition of lifetime as "the time until the first node dies" may be not the best estimate of the actual system lifetime, since the network may still function properly even with fewer nodes. However, as an optimisation objective, the definition is very useful, as maximising the minimum lifetime over all nodes essentially balances the node lifetimes. In the theoretical case that the optimisation system is ideal, this implies that all nodes expire at the same time, which is certainly the end of the network as a whole. If a node dies at run time, the network may reconfigure (see Chapter 6), and the same lifetime metric would again tend to balance the workload across network, and thus maximise its lifetime.

Space. The area that is covered by the nodes, if all nodes would be active, is called the covered area. However, a sensor node only covers the area in its range for a fraction of the time. We therefore introduce the metric *coverage degree* $C^{\mathrm{c}}(c)$. For a point in the covered area, $C^{\mathrm{c}}(c)$ is defined as the percentage of the time that it is covered by *at least one* sensor, during a certain period. The coverage degree for the whole covered area is the minimum coverage degree over the whole area. To calculate $C^{\mathrm{c}}(c)$ for the network/cluster, we would need the locations and sensing ranges, plus the coverage degrees of all the nodes. We could approximate the target area by a grid of points and take the minimum coverage degree for each point. For this example, however, we use the form of (3.4e), which is accurate if every sensor covers some area that cannot be covered by any other sensor.

Finally, a cluster's output traffic and parent node are defined as the output traffic and parent node of its root node.

3.2.3 Model Accuracy

Large model inaccuracies may lead to two different types of problems. Firstly, if the computed metrics are different from the real values, this may result in incorrect conclusions about the

Table 3.3: Model constants for TelosB nodes

Reliability			Time			Space		
k_r	1000		D_{tx}	120	ms	R_{s}	10	m
α	3		D_{s}	35	ms	A	$100 \cdot \lvert\mathcal{N}\rvert$	m^2
d_0	1	m	E_{s}	0.15	mJ	m	$\lvert\mathcal{N}\rvert/100$	
N	-18	dBm	E_{batt}	12	Wh			
b	288	bit	P_{mcu}	5.4	mW			
L	0.015		E_{rx}	8.2	mJ			
			$r_i(n)$ (TT)	$(\lvert c\rvert - 1) \cdot 0.005$	s^{-1}			
			$r_i(n)$ (SM)	$(\lvert c\rvert - 1) \cdot 0.02$	s^{-1}			

Table 3.4: Conversion of transmit power to energy per sent packet for TelosB nodes

P_{tx} (dBm)	E_{tx} (mJ)
-25	3.1
-15	3.6
-10	4.0
-5	5.0
0	6.6

compliance of configurations with constraints. If the constraints are soft, and the deviations small, this may not be a problem. Otherwise, the model should yield conservative metrics, which may turn out better in reality, but not worse, such that constraints are always satisfied. Another potential problem is that wrongly computed metrics may change the dominance order of configurations. Configurations that are computed as being Pareto points, may then actually be dominated by other configurations. In both cases, model inaccuracies may lead to a WSN configuration that is not optimal in terms of the quality metrics.

In case model inaccuracies stem especially from wrongly estimated uncontrollable parameters, which are the constants in the model equations, run-time adaptation may offer a solution. Certain uncontrollables, such as the contention-loss probability and transmission delay, can be measured at run time, or perhaps even the whole mapping function can be adjusted or learnt while the network is operating. Such renewed knowledge can be used in a reconfiguration process to find a better configuration. Chapter 6 explores this idea.

We tested the SM and TT models by simulation of a network of 900 nodes, randomly positioned in an area of 300 × 300 m. The constants in the nodes' mapping functions were chosen to match Crossbow TelosB [16] sensor nodes (the power usage and transceiver parameters). The constants L and D_{tx}, which are actually uncontrollable parameters, were estimated by simulation (it is reasonable to assume that certain constants can be determined empirically at

Table 3.5: Accuracy results for 900-node example network. Differences with simulation in parentheses.

Information Completeness I^c (%)	Detection Speed S^c ($\frac{1}{s} \cdot 10^3$)	Lifetime T^c (h)	Coverage Degree C^c
84 (-3)	41 (-0.4)	3481 (+8)	0.2
63 (-2)	41 (-0.4)	3773 (+120)	0.2
2.0 (+0.5)	41 (-0.4)	4073 (+388)	0.2
78 (-1)	41 (-0.3)	1821 (+3)	0.4
59 (+1)	41 (-0.4)	1970 (+54)	0.4
2.0 (+0.3)	41 (-0.4)	2123 (+195)	0.4
78 (+5)	41 (-0.2)	1214 (-76)	0.6
59 (+6)	41 (-0.1)	1313 (-40)	0.6
2.0 (+0.4)	41 (-0.4)	1415 (+55)	0.6

WSN deployment time). Likewise, the received traffic rate $r_i(n)$ for a node n, is considered to be proportional to the size of the node's sub-tree, where the scale factor is obtained by experiment. The simulations were implemented in the OMNeT++ simulator [64]. The nodes use B-MAC [52] to communicate. See Tables 3.3 and 3.4 for an overview of the used constants.

The method introduced in Chapter 4 was then used to find the Pareto points of the tasks. The quality metrics of the Pareto points for the TT task are shown in Table 3.5. Subsequently, we tested these configurations in a network simulator based on OMNeT++ [64], by configuring the network accordingly. Energy usage, data loss and delay were determined by simulating each configuration, after which the quality-metrics information completeness, detection speed and lifetime were computed (the mapping function for coverage degree is accurate by definition, and therefore not tested). The difference between computed and simulated values is given in brackets in Table 3.5 (e.g. '-10' indicates that the computed value is 10 lower than the simulated value). The deviations are relatively small in general (within 10%). The same tests were done for 18 other random networks of different sizes, for both SM and TT (5 random configurations per network). On average, the deviations were 1.5% (percentage point), 0.9% and 5.4%, for completeness, speed and lifetime respectively. For information-completeness, the deviation is reported in percentage points, because the completeness percentages may become very small (see Table 3.5). In such a case, an acceptable small absolute deviation in percentage points would result in a misleadingly high relative deviation.

3.3 Objectives

Given the definitions of a task and WSN configuration, we specify the following objectives for the task-configuration process. The configuration system should:

1. **At any time, optimise task quality, while meeting all quality and resource constraints.** This means that for a point in time with a given vector of uncontrollable parameters $\bar{u}$, the task should be in a configuration with controllable-parameter vector $\bar{p}$, such that

$$F_{\mathrm{q}}(\bar{p} \cdot \bar{u}) = \min(val(\{F_{\mathrm{q}}(\bar{p}' \cdot \bar{u}) \mid \bar{p}' \in \mathcal{S}_{\mathrm{Pc}}\} \cap \mathcal{D}_{\mathrm{q}})), \tag{3.5}$$

   and

$$F_{\mathrm{r}}(\bar{p} \cdot \bar{u}) \in \mathcal{D}_{\mathrm{r}}. \tag{3.6}$$

   The former requirement means that the quality metrics of the configuration need to satisfy the quality constraints, and they need to be optimal for the *value function* val as well. The value function $val : \mathcal{S}_{\mathrm{Mq}} \rightarrow V$ (often called objective or cost function) is a monotone function that assigns a value in a totally ordered quantity V to a quality-metric vector. Here, val is lifted to sets of metric vectors: $val(\mathcal{C}) = \{val(\bar{c}) \mid \bar{c} \in \mathcal{C}\}$. Note that constraints have priority over value. Furthermore, the latter requirement says that the resource metrics need to satisfy the resource constraints.

2. **Optimise the cost of the configuration process.** We measure the cost in terms of the *total configuration time*, and the *processing* and *communication overhead* per node. The total configuration time is the absolute time spent from the point that configuration is started until all nodes have set the correct parameters. The processing overhead per node is measured in CPU seconds spent, while the communication overhead comprises the amount of data transmitted over the node's radio. For both per-node cost metrics we look at the mean and maximum values across nodes. Finally, the configuration process should scale well to large networks, even though the configuration space is inherently exponentially large (see Section 3.1).

Note that the value function val does not need to be a weighted sum of the metrics. It can be any function that places a total order on the metric space, as long as it is a monotone function. Another example is a function that prioritises some metrics over others (e.g. a higher speed is always more important that a higher lifetime).

Algorithm 3.1: QoS optimisation: one-step method

1 $\mathcal{C} \leftarrow \{\bar{p} \cdot F_{\mathrm{t}}(\bar{p} \cdot \bar{u}) \mid \bar{p} \in \mathcal{S}_{\mathrm{Pc}}|_T\} \triangledown I_{\mathrm{P}}$ compute metrics for given $\bar{u}$, T; hide parameters
2 $\mathcal{C} \leftarrow \mathcal{C} \cap \mathcal{D}_{\mathrm{r}}$ determine metrics that meet resource constraints
3 $\mathcal{C} \leftarrow \mathcal{C} \downarrow I_{\mathrm{Mr}}$ abstract from resource metrics
4 $\mathcal{C}_{\mathrm{opt}} \leftarrow \min(\mathcal{C})$ find Pareto points of quality metrics

We consider the configuration process and the WSN task to be completely decoupled as if they run on separate platforms, such that both of the above objectives do not interfere. Besides the trade-offs between the metrics of a task, there exist high-level *meta* trade-offs between task quality and the cost metrics of the configuration process. There are several choices (meta parameters) in the configuration algorithms that affect this trade-off.

The above objectives are specified for a single task running on a WSN. We may extend it to multiple tasks that share the platform. While this thesis does not cover this case in detail, Section 4.5 gives a brief introduction to the problem.

3.4 Configuration Phases

At any point in time, the configuration system should make sure that a vector $\bar{p}$ of controllables that satisfies (3.5) and (3.6) is installed in the network. Our configuration method splits the optimisation problem in two parts: it first constructs the routing tree – it sets the *parent node* controllable parameter of each node – and subsequently determines values for the remaining controllables based on that tree. This means that we limit the part of the configuration space that we search and therefore may miss out on some potentially good solutions. However, by fixing the tree we are able to view the network as a hierarchy of clusters (see Definition 3.1), such that we can incrementally find *all* Pareto-optimal configurations of the remaining space in a very efficient way.

We distinguish five phases in the configuration process:

1. **Initialisation.** Information about the network needs to be gathered at places where the configuration system needs it. This involves the node locations, node types and capabilities (the controllable-parameter space $\mathcal{S}_{\mathrm{Pc}}$ and mappings to metrics), details about their local environment and state (uncontrollable-parameter space $\mathcal{S}_{\mathrm{Pu}}$ and current vector $\bar{u}$), and the constraints ($\mathcal{D}_{\mathrm{q}}$ and $\mathcal{D}_{\mathrm{r}}$).

2. **Routing-Tree Construction.** The *parent node* controllable parameter of each node needs to be

set to construct a tree T. Properly choosing the tree is important, because it has an impact on both the metrics of the task as well as the performance of the next configuration phase (see Chapter 5). It is therefore a parameter that affects the meta trade-off introduced in Section 3.3.

3. **QoS Optimisation.** Analysis of the remaining configuration space after setting the tree in order to find all resource-constraint satisfying and Pareto-optimal configurations in this space. This phase produces a set of configurations $\mathcal{C}_{\text{opt}}$ according to the program in Algorithm 3.1, where $\mathcal{S}_{\text{Pc}}|_T$ is the controllable-parameter space in which the *parent node* quantities have collapsed to a single value each, according to the tree T, F_{t} is the mapping to task-level metrics, and I_{P} and I_{Mr} are the sets of indices to the controllable-parameter and resource-metric quantities in $\mathcal{C}$ respectively. This program, however, does not scale due to the exponential size of the configuration space. Chapter 4 gives an efficient solution.

4. **Selection.** A configuration that meets the task-level constraints needs to be selected from the set found in the previous phase. Of the configurations that meet the quality constraints, this is the configuration that has the best value according to the value function val. Thus, the selected WSN configuration is $\bar{c}^* = \min(val(\mathcal{C}_{\text{opt}} \cap \mathcal{D}_{\text{q}}))$, and the controllable-parameter vector $\bar{p}^*$ of $\bar{c}^*$ is used to configure the network. The vector $\bar{p}^*$ contains the settings for all nodes in the network. Note that quality constraints can be applied after minimisation (in the previous phase), since the constraints are safe.

5. **Loading.** The nodes need to be informed of the selected configuration, such that they can apply the chosen settings.

The configuration process configures the network for a given situation (set of nodes, $\bar{u}$, constraints, value function). If the situation changes, some of the above phases have to be repeated to arrive at an updated configuration. However, not all the work needs to be redone. Chapter 6 explores ways to reconfigure the network efficiently.

The configuration phases describe a high-level overview of the configuration process, but do not yet make two important decisions: *where* do the computations take place, and what is the *locality* of the algorithms. Computations take place in configuration phases 2, 3, and 4; we assume the initialisation and loading phases need communication only. There are two extremes: all computation is done by a single node (*centralised* computation) versus all nodes – including sensor nodes – do a part of the work (*fully-distributed* computation). For the centralised case, we assume the

compute node is the root of the network, or an external node that is directly connected to the root. Another possibility is to have a number of dedicated configuration nodes scattered throughout the area of deployment. The algorithms introduced in this thesis all come in centralised as well as distributed forms, and both forms have their own benefits and disadvantages.

An important parameter to control the meta trade-off between task quality and configuration efficiency is locality. The best case for task quality is when an algorithm is *globalised*, which means that it has access to information from the whole network and has control over all nodes in the network. At the other end of the spectrum are *localised* algorithms, which involve only the nodes in a certain region around a compute node. In general, only global algorithms can be optimal in terms of task quality, but they are also the most expensive in computation and communication. Especially when reconfigurations are frequently needed, localised algorithms become attractive, which is why they are given special attention in Chapter 6 on adaptation.

3.5 Summary

In this chapter, a precise definition of the configuration space for a WSN is given, involving the network, the task running on the network, parameters and metrics. The networks we consider have a single data sink, and use a routing tree for communication.

Parameters can either be controllable by the configuration system, or uncontrollable (imposed by the environment). Mandatory parameters for the class of networks we consider are the *parent node* (controllable) of a node in the routing tree, and the node *location* (uncontrollable). Metrics appear in two flavours: quality metrics reflect the performance of the task in terms that are relevant for the user, while resource metrics indicate the utilisation of physical resources in the nodes. Parameters and metrics are linked via mappings; a vector of parameters plus the resulting vector of metrics is called a WSN configuration. The size of the configuration space is equal to the number of possible vectors of controllable parameters for the whole WSN. Also defined are constraints on quality and resource metrics.

Quality metrics are specified not only at the task level, but also for groups of nodes called clusters. We view such clusters as hierarchical elements of the network, and define a model component for each level in the hierarchy. A model component for a cluster comprises the parameters of the nodes in the cluster, the metrics of the cluster as a whole, and the mappings between these. The existence of such a hierarchy is essential for the configuration method in Chapter 4.

We further defined hierarchical models for two specific WSN tasks: spatial mapping, in which all nodes periodically take samples that are sent to the user, for instance to determine the temperature profile over the area, and target tracking, in which the objective is to detect and follow target objects. We also assess the accuracy of the models by simulation, and discuss the effects of inaccuracies on the configuration process in general. The simulations show that our models are accurate up to a few percent.

Section 3.3 precisely defined the objectives of the configuration exercise. The objectives are twofold: firstly, quality metrics should be optimised for some value function, while quality and resource constraints should be met, and secondly, the costs of the configuration process itself, in terms of time, processing and communication, should be minimised. As these two objectives are conflicting, there exists a meta trade-off between task quality and configuration cost.

Finally, the configuration process is divided in five phases. The process commences with an initialisation phase, which is followed by a phase in which the routing tree is constructed. Subsequently, the Pareto-optimal WSN configurations are determined, one of these that satisfies the constraints is selected based on the value function, and loaded into the network. These phases are worked out in detail in the following two chapters.

Chapter 4

QoS Optimisation

A crucial step in the configuration process as outlined in Chapter 3 is the QoS-optimisation phase (phase 3). In this phase, we determine the set of Pareto-optimal configurations for the WSN task, given a routing tree. The basic program to compute Pareto points, given in Section 3.4 does not scale. In this chapter, we provide a practical solution that does scale to large networks.

The main section of this chapter is Section 4.1, in which the scalable QoS-analysis is introduced. A detailed overview of the method is given, and its complexity is derived. Next, Section 4.2 discusses optimisations of the algorithm, that are necessary for a practical implementation. The analysis algorithm is initially given as a sequential program for centralised execution, but it can easily be executed in a distributed way as well. Section 4.3 shows how the algorithm can be run on the nodes of the WSN itself. Even though it is efficient in many practical cases, in the worst case, the complexity of the suggested algorithm is still exponential, and thus not scalable. Moreover, when the algorithm is executed directly on the resource-constrained sensor nodes, it may be wise to sacrifice some quality for a more efficient execution of the algorithm. Section 4.4 introduces ways to control the complexity of the algorithm in order to choose a suitable point in the quality/configuration-cost trade-off space. Subsequently, Section 4.5 provides ideas on how to map multiple tasks together on a single WSN, and experimental results that verify the scalability and other aspects of the algorithm are given in Section 4.6.

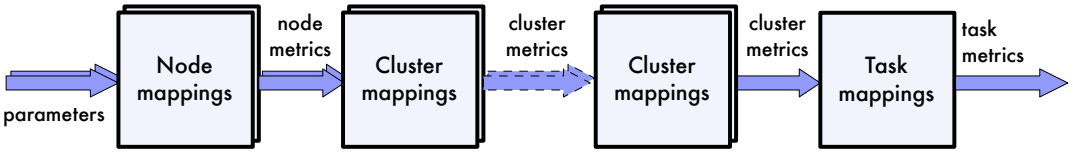


Figure 4.1: A hierarchical model of parameters, metrics and incremental mappings.

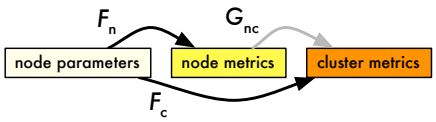


Figure 4.2: Cluster-level quality metrics can be derived from the parameters of the nodes in the cluster by a mapping F_{c}. Alternatively, they can be derived from node-level quality metrics (G_{nc}).

4.1 A Scalable Approach

To find the set of Pareto-optimal configurations using a flat system model, we would have to map *every* parameter vector in $\mathcal{S}_{\mathrm{Pc}}|_T$ to a vector of metrics by means of task-level mapping functions, apply constraints, and Pareto minimise the resulting set of quality metrics. For the target-tracking and spatial-mapping tasks, the task mappings are defined by the combination of the functions in Tables 3.1 and 3.2. However, the size of $\mathcal{S}_{\mathrm{Pc}}|_T$ increases exponentially with the number of nodes in the network, so such an approach would not be scalable. We consider the procedure to find the Pareto points *scalable*, if the run time of the algorithm does not grow faster than linearly with the size of the network (the number of nodes).

As explained in Section 3.1.2, we use hierarchical models, where nodes form the lowest level, clusters form intermediate levels, and highest level is the task running at the network. The models at each level have the same structure (see Figure 3.1): they are mappings from parameters to metrics. Figure 4.1 illustrates this structure. This hierarchy can be exploited, by building and minimising sets of configurations step by step: start with individual nodes, and then incrementally combine nodes into clusters, until the task level is reached. This solution may be seen as an example of dynamic programming, as we optimally solve sub-problems, which are then (again optimally) joined together. We expect that minimisation discards a significant number of dominated configurations in each step, such that only the resulting (Pareto-optimal) configurations need to be kept for the next level. The correctness of this approach – the resulting set of configurations from the one-step and clustered approaches should be identical – and its complexity are investigated in this section.

4.1.1 Overview of the Cluster Method

The first step in the cluster method is to find the Pareto points for each node as one-node clusters. We skip the node metrics and straightaway use cluster metrics, essentially merging the first two boxes in Figure 4.1. We denote the controllable-parameter space of a node i by $\mathcal{S}_{\mathrm{Pc},i}|_T$, so for an n-node network, $\mathcal{S}_{\mathrm{Pc}}|_T = \mathcal{S}_{\mathrm{Pc},0}|_T \times \mathcal{S}_{\mathrm{Pc},1}|_T \times \ldots \times \mathcal{S}_{\mathrm{Pc},n-1}|_T$. This space maps to a cluster-level

metric space by a mapping $F_{\mathrm{c}} : \mathcal{S}_{\mathrm{Pc},i}|_T \rightarrow \mathcal{S}_{\mathrm{M},i}$. In the target-tracking and spatial-mapping examples, the mapping is specified by the combination of the mappings F_{n} (Table 3.1) and G_{nc} (Table 3.2): $F_{\mathrm{c}} = G_{\mathrm{nc}} \circ F_{\mathrm{n}}$ (see Figure 4.2). From the resulting set of metric vectors, only the ones that meet the resource constraints are kept, and the remaining set is minimised on quality metrics. Hence, the set of Pareto-optimal node-level configurations of a node i is calculated as in Algorithm 4.1. In this algorithm, $\mathcal{D}_{\mathrm{r},i}$ is the set of configurations for node i that satisfy the resource constraints. I_{P}, I_{Mr}, and I_{Mq} are sets of indices to the parameter, resource-metric, and quality-metric quantities in the configurations respectively; we assume that these sets are automatically updated to contain the right indices after any operation.

Note that the procedure in Algorithm 4.1 is similar to the basic algorithm of Section 3.4, but then for a single node instead of the whole task, as those are only specified at the task level. We assume that resource constraints are applied at the node level, as they reflect physical limitations. While they can just be applied in the final step (at task level), it is more efficient to use resource constraints at each step of the algorithm, as it leads to a further reduction of the size of parameter sets. The result of Algorithm 4.1 is that the initial set of node parameter vectors is pruned by the minimise and constraint operations. Only the remaining parameter vectors are considered in the next step, in which clusters are combined.

The basic form of the cluster method is shown as Algorithm 4.2. The main loop of the algorithm in lines 4–11 incrementally combines clusters. The configuration sets of the clusters are stored as variables $\mathcal{C}_i$, where i is the index of the root node of the cluster. Before and after each loop iteration, these configuration sets are equal to the Pareto-optimal configurations in the product of the parameter sets of the nodes contained in the cluster. This invariant is initialised in lines 1–2, in which the function `CreateOneNode` of Algorithm 4.1 is used to prune and store the parameter sets for each node. In each iteration, two or more clusters are chosen to be combined (line 6). This choice is very important for the correctness of the algorithm, as will become clear in Section 4.1.2. The cluster step further involves putting the parameter sets together (line 7), deriving metrics with the mapping F_{c} in which we assume only quality metrics are computed since resources only play a role at node level (line 8), and minimisation (line 9). Subsequently, the cluster metrics are removed, as they are no longer necessary. The clustering loop terminates when all nodes are in a single cluster. In our examples of Section 3.2, the resulting cluster metrics are also the task-level metrics; if not, the task-level metrics are derived by a mapping F_{t} (line 13). The algorithm terminates after minimising the resulting set (line 14). Note that in each step of this

Algorithm 4.1: Creation of a one-node cluster

```
function CreateOneNode(i):
    C_i ← S_{Pc,i}|_T                          initial set of parameter vectors for node i
    C_i ← {p̄ · F_c(p̄ · ū) | p̄ ∈ C_i} ▽ I_P    add derived metrics and hide parameters
    C_i ← (C_i ∩ D_{r,i}) ↓ I_Mr               constrain and abstract from resource metrics
    C_i ← min(C_i)                             minimise on quality metrics
    C_i ← (C_i ↓ I_Mq) △ I_P                   remove quality metrics and unhide parameters
    return C_i
```

Algorithm 4.2: Computing task-level Pareto points by combining clusters incrementally

```
for all nodes i ∈ N:
    C_i ← CreateOneNode(i)                     create one-node cluster set for node i

Cl ← N                                         initial set of clusters: all nodes
while |Cl| > 1:                                repeat until a single cluster remains
    S ← remove sub-set from Cl                 choose indices of clusters to combine
    C_prod ← ∏_{j∈S} C_j                       create product set
    C_prod ← {p̄ · F_c(p̄ · ū) | p̄ ∈ C_prod} ▽ I_P   derive cluster metrics and hide parameters
    C_prod ← min(C_prod)                       minimise quality metrics
    C_{rt(S)} ← (C_prod ↓ I_Mq) △ I_P          remove metrics and unhide parameters
    Cl ← Cl ∪ {rt(S)}                          update set of clusters
                                               (rt gives the root node of a cluster)

C ← {m̄ · F_t(m̄ · ū) | m̄ ∈ C_0} ▽ I_P          derive task metrics and hide parameters
C_opt ← min(C)                                 obtain task-level Pareto points
```

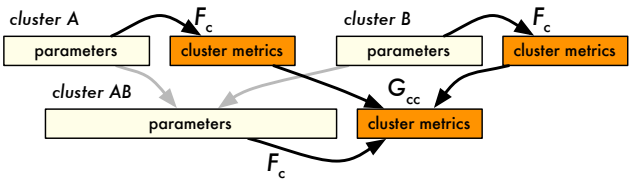


Figure 4.3: Clusters A and B are combined into cluster AB. The cluster quality metrics of AB can always be directly derived from the parameters of the nodes in the cluster (F_c). If they can also be derived from the cluster-level metrics of A and B and this mapping G_{cc} is monotone, the combining action is also monotone.

algorithm, again, the sets of parameter vectors for each node are pruned, and only the remaining parameter vectors are used in the following steps.

Clusters can be combined into a larger cluster by constructing the free product of the parameter sets of those clusters, and deriving new cluster quality metrics from the parameters with F_c as done in Algorithm 4.2. However, it is sometimes possible to *incrementally* derive new metrics from the *metrics* of the clusters that are combined instead (see Figure 4.3). Such an incremental mapping is usually more efficient in terms of computation and storage needs.

Definition 4.1 (Incremental mapping). An *incremental* mapping is a mapping $G : \mathcal{S}_{M,1} \to \mathcal{S}_{M,2}$ that derives a vector of metrics from another vector of (possibly different) metrics. The function G can be lifted to sets: $G(\mathcal{C}) = \{G(\bar{c}) \mid \bar{c} \in \mathcal{C}\}$, with $\mathcal{C} \subseteq \mathcal{S}_{M,1}$.

We use the letter F for mappings from parameters, and the letter G for incremental mappings. For a mapping F, we use subscripts 'n', 'c' and 't' for a mapping to node-, cluster- and task-level metrics respectively (as done above). For G, we use a two-letter subscript, to indicate the source and destination level. One example of an incremental mapping has already been used: the mapping G_{nc}, from node metrics to cluster metrics (Table 3.2). Figure 4.3 shows the two ways of computing metrics of the new cluster after combining two clusters: first combining the parameters and then applying F_c, or directly applying G_{cc} to the cluster metrics of the two clusters (the mapping G_{cc} for the example tasks is given later). The mapping G_{cc} is a tuple $(g_0, g_1, \ldots, g_{k-1})$ of k mapping functions for k metrics.

Node-level parameters, metrics and mappings can be different for any node, but to make combining clusters easier, it is convenient to have the same cluster-level metrics for all clusters. The result is that we need only one type of cluster-to-cluster mapping G_{cc}. In the remainder of this thesis, without loss of generality, we assume that this is the case. If certain groups of nodes need different metrics, the cluster metrics used should then be the union of all metrics needed in the network. See Section 4.5 for an example of this. If the cluster-metric space is given by $\mathcal{S}_M$

for all clusters, and ℓ clusters are being combined, the mapping G_{cc} is defined as $(\mathcal{S}_{\text{M}})^{\ell} \rightarrow \mathcal{S}_{\text{M}}$. This means that for clusters with k metrics, G_{cc} is a tuple of k mapping functions on vectors of $k \cdot \ell$ metric values. This number is bounded if the number of clusters that are simultaneously combined is bounded. The mapping functions in F_{c}, on the other hand, operate on vectors of parameter values for all nodes in the combined cluster, which is a number that grows with the size of the clusters. This implies that incremental mappings are more efficient. Moreover, Section 4.2.3 shows how we can also make use of this to significantly reduce the memory demands of the algorithm.

4.1.2 Monotonicity

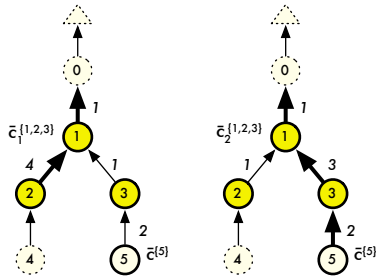
It is generally not possible to combine any arbitrary groups of nodes as clusters. Firstly, all mapping functions to cluster metrics need to be defined for the compound cluster. We are targeting networks that use a routing tree, and some metrics – such as *information completeness* in our example – may be defined with respect to this tree. Therefore, each cluster needs to form a tree: a sub-tree of the network's routing tree, as in Definition 3.1.

Secondly, we need to check for *monotonicity*: we need to make sure that the configurations that are removed by minimisation in earlier clustering steps, could never become optimal in later steps. This concept is related to monotonicity for mapping functions as defined in Definition 2.2.

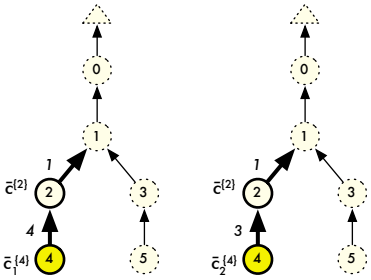
Definition 4.2 (Monotonicity of a clustering step). Suppose we are combining ℓ clusters having parameter sets $\mathcal{C}_i$ and $0 \leq i < \ell$. The action of combining these clusters is monotone, iff for all $\bar{c}_1^i, \bar{c}_2^i \in \mathcal{C}_i$ and $0 \leq i < \ell$, $F_{\text{c}}(\bar{c}_1^i) \preceq F_{\text{c}}(\bar{c}_2^i)$ implies $F_{\text{c}}(\bar{c}_1^0 \cdot \ldots \cdot \bar{c}_1^{\ell-1}) \preceq F_{\text{c}}(\bar{c}_2^0 \cdot \ldots \cdot \bar{c}_2^{\ell-1})$.

A monotone clustering step preserves the dominance order of cluster configurations. This implies that a cluster configuration $\bar{c}$ that is dominated before the clustering step, can safely be removed by minimisation, because all configurations of the combined cluster that incorporate $\bar{c}$ would be dominated by other configurations after clustering. In other words, *if all clustering steps are monotone, none of the eventual task-level Pareto points are lost in the incremental algorithm*, and the result of the incremental algorithm is the same as the result from the all-at-once algorithm.

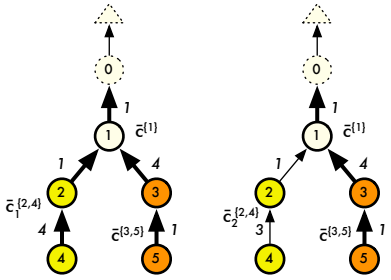
Lemma 4.1. *A clustering step in which clusters* 0 *to* $\ell - 1$ *are combined is monotone, if the mapping* F_{c} *to calculate the metrics of the combined cluster can be rewritten as a monotone incremental mapping* G_{cc} *from only*



(a) Non-monotone step



(b) Monotone step 1



(c) Monotone step 3

Figure 4.4: Examples of non-monotone (a) and monotone clustering steps (b)–(c). A number at the arc coming out of node i is the delay T_i.

the cluster-level metrics of clusters that are combined (parameters of individual nodes are not explicitly needed):

$$F_{\mathrm{c}}\left(\bar{c}^0 \cdot \ldots \cdot \bar{c}^{\ell-1}\right) = G_{\mathrm{cc}}\left(F_{\mathrm{c}}(\bar{c}^0) \cdot \ldots \cdot F_{\mathrm{c}}(\bar{c}^{\ell-1})\right) \tag{4.1}$$

for some monotone function $G_{\mathrm{cc}} : (\mathcal{S}_M)^\ell \to \mathcal{S}_M$. *For k metrics, the incremental mapping G_{cc} must be a tuple* $(g_0, g_1, \ldots, g_{k-1})$ *of k* monotone *functions.*

Proof. This follows immediately from the monotonicity of G_{cc}. □

This result means that the availability of an incremental mapping for cluster metrics does not only lead to more efficient computations in many cases, it also ensures the monotonicity of the cluster algorithm. Note that for monotonicity to hold, it is not needed to actually *use* the incremental mapping; it just needs to exist. However, monotonicity is only guaranteed if the incremental mapping functions are monotone (non-decreasing). New cluster-level metrics can be derived by the function F_{c} from the parameters of all *nodes* inside the new cluster, by the function G_{cc} from the metrics of all the *clusters*, or a hybrid form. This is an implementation choice that can be made per mapping function, based on efficiency of computation and storage. Before providing G_{cc} for the example models of Section 3.2, we first illustrate the monotonicity of the cluster method by means of an example.

4.1.3 A Monotone Clustering Order

Consider a mapping function f_{d} to calculate a maximum-delay quality metric (captured in quantity Q_{d}) for a cluster, given a parameter vector $\bar{c}$. Note that the WSN model for target tracking has a speed metric instead of a delay metric. For the sake of the example, however, we use the delay, for which lower values are preferred ($\preceq_{Q_{\mathrm{d}}}$ equals $\leq$). We express f_{d} in terms of parameter values T_i, which indicate the time to send a message from node i to its parent in the routing tree.

$$f_{\mathrm{d}}(\bar{c}) = \max_{p \in L} \left\{ \sum_{i \,\mathrm{on}\, p} T_i \right\}, \tag{4.2}$$

where L is the set of paths from all leaf nodes in the cluster to the root node (including the root node itself). Clearly, this function does not only depend on the T_i values of the nodes in the cluster; also the way the nodes are connected in the routing tree plays an important role. Therefore, when combining clusters with maximum delay as a quality metric, it is necessary that these clusters are connected by links in the network's routing tree, such that the new cluster is a tree. Furthermore, we make sure that the clustering step is monotone by ensuring that (4.1) holds for f_{d}. We first give an example of a clustering strategy that fails monotonicity condition. Subsequently, we show a clustering strategy that is monotone. Throughout the example, we only show the delay metric, but keep in mind that there may be other quality metrics as well. This means that even if a configuration has a worse delay than another, it may be a Pareto point, depending on the values of other metrics. For convenience, we write a configuration $\bar{c}$ as a vector of only T_i values.

Suppose we have a cluster $\{1, 2, 3\}$, as in Figure 4.4(a). The cluster has multiple possible configurations (different values of T_1, T_2 and T_3). The figure shows two configurations: $\bar{c}_1^{\{1,2,3\}} = (1, 4, 1)$ and $\bar{c}_2^{\{1,2,3\}} = (1, 1, 3)$. The cluster delays are $f_{\mathrm{d}}(\bar{c}_1^{\{1,2,3\}}) = \max\{1 + 4, 1 + 1\} = 5$ and $f_{\mathrm{d}}(\bar{c}_2^{\{1,2,3\}}) = \max\{1 + 1, 1 + 3\} = 4$ (the thick arrows in the figure show the bottleneck path). Thus, $f_{\mathrm{d}}(\bar{c}_2^{\{1,2,3\}}) \preceq_{Q_{\mathrm{d}}} f_{\mathrm{d}}(\bar{c}_1^{\{1,2,3\}})$ and after minimisation, $\bar{c}_1^{\{1,2,3\}}$ may be eliminated (depending on the other metrics). Now we join this cluster with downstream cluster $\{5\}$, with one configuration $\bar{c}^{\{5\}} = (2)$. The new configurations are $\bar{c}_1^{\{1,2,3\}} \cdot \bar{c}^{\{5\}} = (1, 4, 1, 2)$ and $\bar{c}_2^{\{1,2,3\}} \cdot \bar{c}^{\{5\}} = (1, 1, 3, 2)$. The delays become $f_{\mathrm{d}}(\bar{c}_1^{\{1,2,3\}} \cdot \bar{c}^{\{5\}}) = \max\{1 + 4, 1 + 1 + 2\} = 5$ and $f_{\mathrm{d}}(\bar{c}_2^{\{1,2,3\}} \cdot \bar{c}^{\{5\}}) = \max\{1 + 1, 1 + 3 + 2\} = 6$. This implies that $f_{\mathrm{d}}(\bar{c}_1^{\{1,2,3\}} \cdot \bar{c}^{\{5\}}) \preceq_{Q_{\mathrm{d}}} f_{\mathrm{d}}(\bar{c}_2^{\{1,2,3\}} \cdot \bar{c}^{\{5\}})$, but $\bar{c}_1^{\{1,2,3\}}$ may have been discarded in the previous step! Therefore, this clustering step is non-monotone. The reason is that the addition of a *downstream* cluster extends the bottleneck path in one configuration but not in the other one. Observe that this cannot occur

when adding upstream clusters: the bottleneck path is then always affected.

Now we use a clustering order that is monotone. Figure 4.4(b) shows a possible first clustering step, after initialisation. Here, one-node clusters $\{2\}$ and $\{4\}$ are combined. The figure shows two configurations $\bar{c}_1^{\{4\}} = (4)$ and $\bar{c}_2^{\{4\}} = (3)$ of cluster $\{4\}$, and one configuration $\bar{c}^{\{2\}} = (1)$ of cluster $\{2\}$. The mapping function for the combined cluster $\{2, 4\}$ is simply $T_4 + T_2$, which is 5 and 4 for configurations $\bar{c}_1^{\{4\}} \cdot \bar{c}^{\{2\}}$ and $\bar{c}_2^{\{4\}} \cdot \bar{c}^{\{2\}}$ respectively. This step is clearly monotone, and this is always the case when combining clusters that contain only one path: the max-function can be left out and the remaining summation is always monotone.

The second clustering step in the example would be the combination of $\{3\}$ and $\{5\}$, which goes in the same way as step 1. A possible third step is given in Figure 4.4(c). In this step, we are joining cluster $\{2, 4\}$ (having configurations $\bar{c}_1^{\{2,4\}} = (1, 4)$ and $\bar{c}_2^{\{2,4\}} = (1, 3)$) with $\{3, 5\}$ (configuration $(4, 1)$) and $\{1\}$ (configuration (1)). Configuration $\bar{c}_1^{\{2,4\}}$ has a longer delay ($1 + 4 = 5$) than $\bar{c}_2^{\{2,4\}}$ ($1 + 3 = 4$), so it could have been discarded in step 1. Therefore, it should not be possible that a combination of this configuration with any of the other clusters' configurations becomes a unique Pareto point. The joint cluster's delay is

$$\begin{aligned} f_{\mathrm{d}}\left(\bar{c}^{\{2,4\}} \cdot \bar{c}^{\{3,5\}} \cdot \bar{c}^{\{1\}}\right) &= \max\left\{T_4 + T_2 + T_1,\ T_5 + T_3 + T_1\right\} \\ &= \max\left\{f_{\mathrm{d}}(\bar{c}^{\{2,4\}}) + f_{\mathrm{d}}(\bar{c}^{\{1\}}),\ f_{\mathrm{d}}(\bar{c}^{\{3,5\}}) + f_{\mathrm{d}}(\bar{c}^{\{1\}})\right\} \\ &= \max\left\{f_{\mathrm{d}}(\bar{c}^{\{2,4\}}),\ f_{\mathrm{d}}(\bar{c}^{\{3,5\}})\right\} + f_{\mathrm{d}}(\bar{c}^{\{1\}}). \end{aligned} \tag{4.3}$$

The last line can be seen as a function $g(x, y, z) = \max\{x, y\} + z$, which is monotone, and therefore this clustering step is monotone according to Lemma 4.1. In the example, using $\bar{c}_1^{\{2,4\}}$ or $\bar{c}_2^{\{2,4\}}$ will lead to a combined-cluster delay of $1+4+1 = 6$ or $4+1+4 = 6$ respectively. Although $\bar{c}_1^{\{1,2,3,4,5\}}$ is not strictly worse than $\bar{c}_2^{\{1,2,3,4,5\}}$, it is still dominated (equal values dominate each other), so $\bar{c}_1^{\{2,4\}}$ could have been safely removed. Configuration $\bar{c}_1^{\{1,2,3,4,5\}}$ may be worse in dimensions other than Q_{d} to render it strictly dominated, or the two configurations may be identical in terms of metrics, in which case only one needs to be kept. But $\bar{c}_1^{\{1,2,3,4,5\}}$ will never strictly dominate $\bar{c}_2^{\{1,2,3,4,5\}}$.

We also see in (4.3) how the delay mapping function for step 3 can be rewritten from a function that operates on parameters to an incremental mapping function on cluster-level metrics. The implementation of the latter is clearly the most efficient, since the computation is easier and fewer values need to be stored.

The final step to complete the network is to combine cluster $\{1,2,3,4,5\}$ with cluster $\{0\}$, assuming for example $T_0 = 2$. This step is straightforward: the delay is equal to $f_{\mathrm{d}}\left(\bar{c}^{\{1,2,3,4,5\}} \cdot \bar{c}^{\{0\}}\right) = f_{\mathrm{d}}\left(\bar{c}^{\{1,2,3,4,5\}}\right) + f_{\mathrm{d}}\left(\bar{c}^{\{0\}}\right)$ and this step is thus monotone. The delays are $6 + 2 = 8$ for both configurations.

Definition 4.3 (Leaf Cluster). A *leaf cluster* of a WSN is cluster with the special property that for each node in the cluster, all its descendants in the WSN's routing tree are also included in the cluster.

Proposition 4.2 (Monotonocity of Algorithm 4.2 for f_{d}). *All clustering steps of Algorithm 4.2 are monotone for f_{d}, if in each clustering step, the clusters that are combined form a tree with root node R that (besides R) comprises all R's descendants in the network's routing tree. That is, each newly formed cluster is a leaf cluster.*

Proof. For each step of the algorithm, we need to ensure that f_{d} is defined for the compound cluster, and that the step is monotone. The algorithm maintains two invariants:

1. Each cluster is a tree. This ensures that f_{d} is defined for each cluster.
2. A cluster either contains exactly one node or it is a leaf cluster.

Line 4 initialise both invariants, while line 6 plus the selection condition trivially maintains them, whichever cluster is chosen for combination. Invariant 2 ensures that, when combining clusters, the root of a multiple-node cluster M is always connected to the root R of the new cluster via a path containing only one-node clusters. If not, a node belonging to another multiple-node cluster would be on this path, but this implies the existence of a multiple-node cluster that is not a leaf cluster, which contradicts invariant 2. The maximum delay from any leaf node in cluster M to R is equal to the maximum delay of cluster M as a whole, plus the delays of the extra nodes on the path (including R). This is a summation of only cluster metrics, which is monotone. Thus, function f_{d} applied to a combination of one-node and multiple-node clusters is equivalent to the maximum over the maximum delays for all leaf clusters (either multiple-node or one-node). This is a monotone function using only metrics of the compound clusters, which ensures that the clustering step itself is monotone for the delay metric (by Lemma 4.1). □

We reorganise Algorithm 4.2 into Algorithm 4.3, a recursive form that uses incremental mappings. The recursive function `CreateCompound` should be called with the index of the network's root node as argument (assumed to be 0, line 19). This algorithm enforces a monotone

Algorithm 4.3: Monotone cluster combining with incremental mappings

```
1  function CreateOneNode(i):
2      C_i ← S_Pc,i|_T                          initial set of parameter vectors for node i
3      C_i ← {p̄ · F_c(p̄ · ū) | p̄ ∈ C_i} ▽ I_P   append derived metrics and hide parameters
4      C_i ← (C_i ∩ D_r,i) ↓ I_Mr                constrain and abstract from resource metrics
5      C_i ← min(C_i)                           minimise on quality metrics
6      return C_i
7
8  function CreateCompound(i):
9      C_prod ← CreateOneNode(i)                create one-node cluster set for root node i
10     if i is a leaf node:
11         return C_prod                        return one-node cluster set if i is a leaf
12     for each child j of i:                   recursively create product set
13         C_prod ← C_prod × CreateCompound(j)
14     C_prod ← {c̄ · G_cc(c̄) | c̄ ∈ C_prod}      append derived quality metrics
15     C_prod ← C_prod ↓ I_Mlow                  abstract from lower-level metrics
16     C_prod ← min(C_prod)                     minimise on quality metrics
17     return C_prod                            return compound cluster's Pareto set
18
19 C_clus ← CreateCompound(0)                   cluster-level Pareto points for network
20 C ← {c̄ · G_ct(c̄) | c̄ ∈ C_clus}               derive task quality metrics
21 C ← C ↓ I_Mlow                               abstract from lower metrics
22 C_opt ← min(C)                               task-level Pareto points
```

Table 4.1: Incremental mappings (G_{cc}) for a cluster c

Reliability

Inf. completeness
$$I^{\mathrm{c}}_{\Sigma}(c) = I^{\mathrm{c}}_{\Sigma}\left(rt(c)\right) \cdot \left(1 + \sum_{i \in ch(c)} I^{\mathrm{c}}_{\Sigma}(i)\right) \tag{4.4a}$$

Time

Reporting rate (SM)
$$r^{\mathrm{c}}_{\Sigma}(c) = \sum_{i \in sub(c)} r^{\mathrm{c}}_{\Sigma}(i) \tag{4.4b}$$

Detection speed (TT)
$$S^{\mathrm{c}}(c) = \min\left(S^{\mathrm{c}}(rt(c)), \min_{i \in ch(c)} \frac{1}{S^{\mathrm{c}}(i)^{-1} + D_{\mathrm{tx}}}\right) \tag{4.4c}$$

Lifetime
$$T^{\mathrm{c}}(c) = \min_{i \in sub(c)} T^{\mathrm{c}}(i) \tag{4.4d}$$

Space

Coverage degree
$$C^{\mathrm{c}}(c) = \min_{i \in sub(c)} C^{\mathrm{c}}(i) \tag{4.4e}$$

Note: (4.4a) *and* (4.4c) *depend on a tree; the others do not. For combined cluster* c*, the root cluster is denoted* $rt(c)$*, the set of child clusters* $ch(c)$*;* $sub(c) = \{rt(c)\} \cup ch(c)$.

clustering order: it starts at the leaf nodes and continues up towards the root, and clusters are formed as leaf clusters in compliance with Proposition 4.2. The function `CreateOneNode` is slightly modified for use in this version of the algorithm: line 6 in Algorithm 4.1 is omitted, such that quality metrics are left in the configurations, and parameters remain to be hidden. Line 14 of Algorithm 4.3 uses the incremental mapping G_{cc} to compute the quality metrics for the compound cluster from the metrics of the clusters that are contained in the compound. The lower-level metrics are then no longer needed, and therefore removed in line 15. The result is a set of Pareto-optimal configurations for the cluster containing all nodes. If needed, task-level metrics can be derived from these (line 20), and finally, the result is minimised (line 22). Note that, while the parameters for each node in the resulting set are hidden and do not play a role in the computations (except at the node level), they are still part of the configurations.

4.1.4 Correctness of Example Models

To prove the correctness of the cluster method for the WSN models of Section 3.2, we need to provide a monotone incremental mapping G_{cc} (Lemma 4.1). Monotonicity should be verified

for each incremental mapping function g_i separately, and a clustering step should only combine clusters that can be monotonically combined (note that the mapping functions in mappings F *need not* be monotone). Table 4.1 gives G_{cc}. I^{c}_{Σ} and r^{c}_{Σ} are cumulative metrics, as indicated by the sub-script Σ, that should be divided by $|c|$ to get the actual quality metrics I^{c} and r^{c} of Table 3.2. The incremental computation of these cumulative metrics is more efficient than the computation of the actual metrics, as we leave out the divide operation.

Theorem 4.3 (Monotonicity of Algorithm 4.3). *The cluster method (Algorithm 4.3) is monotone for the WSN models of Section 3.2.*

Proof. It can be shown by similar arguments as in Proposition 4.2 (plus the fact that minimum, addition and multiplication are monotone) that, given the clustering strategy in Algorithm 4.3, all mapping functions in the model can be written as monotone functions from cluster to cluster metrics, as in Table 4.1. Therefore, by Lemma 4.1, Algorithm 4.3 is monotone for the WSN model. □

4.1.5 Complexity

The operations of Pareto algebra mostly have a polynomial time complexity which is at most quadratic (for the Simple Cull minimisation algorithm) in the number of configurations n [22]. A crucial operation is combining two sets of configurations of size n and m by a free product, which has complexity $\mathcal{O}(n \cdot m)$, and increases the number of configurations from $n + m$ to $n \cdot m$. The free product increases the number of configurations, while minimisation and applying constraints never increase, and usually reduce this number.

The efficiency of Algorithms 4.2 and 4.3 mainly depends on the number of clusters ℓ that are combined per step, and the number of configurations $|\mathcal{C}_i|$ in each cluster i. Line 7 of Algorithm 4.2 (lines 12–13 of Algorithm 4.3) combines configuration sets with a free product. The size of the resulting set $\mathcal{C}_{\text{prod}}$ is equal to $|\mathcal{C}_0| \cdot \ldots \cdot |\mathcal{C}_{\ell-1}|$, and the time complexity of the free-product operation is $\mathcal{O}(|\mathcal{C}_{\text{prod}}|)$. The complexity of the derivation step in the following line also depends on $|\mathcal{C}_{\text{prod}}|$ (metrics need to be derived for each new configuration), but on the complexity of the mapping functions as well. Finally, the complexity of the minimisation operation also depends on $|\mathcal{C}_{\text{prod}}|$, as said above.

If we consider the number of configurations per node as given and at most n, mapping functions can be evaluated in constant time, and assuming a quadratic minimisation algorithm is

used, the complexity of joining all nodes in one step is $\mathcal{O}(n^{2|\mathcal{N}|})$. This is obviously not scalable and therefore not useful for WSNs in general. Therefore, it makes sense to join as few clusters as possible in each step and to rely on minimisation to keep the configuration sets small. The algorithm's complexity could even become linear, if each step is able to reduce the set size to roughly m when cluster configuration sets of size m are combined. On the other hand, if the minimisation operation does not manage to significantly reduce the size of $\mathcal{C}_{\text{prod}}$, the complexity would again be exponential. This number of configurations that can be minimised away greatly depends on the mapping functions and the configurations' values, and it is hard to give general bounds, as we do not make any assumptions about these values and the precise mappings (besides the monotonicity requirement). If hardly any configurations are dominated in each step, the run time still grows exponentially with the number of nodes. However, this is a very unlikely case; experiments show that, in practice, only very few configurations are Pareto optimal, and the run time of the cluster algorithm scales approximately linearly with the number of nodes in the network (see Section 4.6). Furthermore, in such practical cases, we will always find *all* Pareto points. Nevertheless, we want to make the worst-case behaviour tractable. We can do so by limiting the number of configurations in the quality-metric space. The reduction method introduced in Section 4.4 works out the details. Limiting $|\mathcal{C}_{\text{prod}}|$ enforces an upper bound on the run time, in exchange for lower-quality results. This provides a means to control practical complexity, and is especially useful when running the algorithm on sensor nodes.

Furthermore, it follows from the significance of $\mathcal{C}_{\text{prod}}$ that, to keep the run time low, it is desirable to combine as few clusters per step as possible. However, due to the restriction imposed by the monotonicity condition on the clusters that may be combined, a given (one-node) root cluster is always combined with all of its child clusters at the same time. The number of configuration sets that are combined in a cluster step is thus equal to the *node degree* (number of child nodes) plus one. Therefore, the node degree of nodes in the routing tree is crucial, and should be as low as possible. We come back to this in Chapter 5, in which tree-construction algorithms with node-degree reduction are suggested.

4.2 Implementation

To efficiently implement Algorithm 4.3, we suggest optimisations that significantly improve its memory complexity and therefore also its timing. We give some further implementation details as well. The optimised algorithm is given as Algorithm 4.4.

Algorithm 4.4: Optimised implementation of Cluster algorithm

1 **function** `CreateOneNode(i):`
2 $\quad \mathcal{C}_i \leftarrow \mathcal{S}_{\mathrm{Pc},i}|_T$ — initial set of parameter vectors for node i
3 $\quad \mathcal{C}_i \leftarrow \{(j) \cdot \mathcal{C}_i\,[j] \mid 0 \le j < |\mathcal{C}_i|\} \triangledown 0$ — prepend and hide index
4 $\quad \mathcal{C}_i \leftarrow \{\bar{p} \cdot F_{\mathrm{c}}(\bar{p} \cdot \bar{u}) \mid \bar{p} \in \mathcal{C}_i\} \downarrow I_{\mathrm{P}}$ — append metrics, remove parameters
5 $\quad \mathcal{C}_i \leftarrow (\mathcal{C}_i \cap \mathcal{D}_{\mathrm{r},i}) \downarrow I_{\mathrm{Mr}}$ — constrain and abstract from resources
6 $\quad \mathcal{C}_i \leftarrow \min(\mathcal{C}_i)$ — minimise on quality metrics
7 $\quad$ **return** $\mathcal{C}_i$
8
9 **function** `CreateCompound(`i`):`
10 $\quad \mathcal{C} \leftarrow$ `CreateOneNode(`i`)` — create one-node cluster set for root node i
11 $\quad$ **if** i **is a leaf node:**
12 $\quad\quad$ **return** $\mathcal{C}$ — return one-node cluster set if i is a leaf
13 $\quad$ **add** $\mathcal{C}$ **to list S** — initialise list of clusters
14 $\quad$ **for each child** j **of** i**:** — recursively create child clusters
15 $\quad\quad$ `add CreateCompound(`j`) to S`
16 $\quad \mathcal{C}_{\min} \leftarrow \varnothing$ — initialise minimal set
17 $\quad$ **for all** $\bar{c}$ **in Product(S):** — iterate through product set
18 $\quad\quad \bar{c} \leftarrow \bar{c} \cdot G_{\mathrm{cc}}(\bar{c})$ — append derived quality metrics
19 $\quad\quad \bar{c} \leftarrow \bar{c} \downarrow I_{\mathrm{Mlow}}$ — abstract from lower-level quantities
20 $\quad\quad \bar{c} \leftarrow (\bar{c} \cdot R_{\bar{q}}(\bar{c})) \triangledown I_{\mathrm{Mq}}$ — add quantised metrics; hide original
21 $\quad\quad \mathcal{C}_{\min} \leftarrow$ `AddAndMin(`$\mathcal{C}_{\min}$`,` $\bar{c}$`)` — add to set and keep Pareto minimal
22 $\quad$ `Index[`i`]` $\leftarrow (\mathcal{C}_{\min} \vartriangle I_{\mathrm{I}}) \downarrow (I_{\mathrm{Mq}} \cup I_{\mathrm{Q}})$ — extract indexing table
23 $\quad \mathcal{C}_i \leftarrow (\mathcal{C}_{\min} \vartriangle I_{\mathrm{Mq}}) \downarrow (I_{\mathrm{I}} \cup I_{\mathrm{Q}})$ — extract unquantised metrics
24 $\quad \mathcal{C}_i \leftarrow \{(j) \cdot \mathcal{C}_i\,[j] \mid 0 \le j < |\mathcal{C}_i|\} \triangledown 0$ — prepend and hide index
25 $\quad$ **return** $\mathcal{C}_i$ — return compound cluster's Pareto set
26
27 $\mathcal{C}_{\mathrm{clus}} \leftarrow$ `CreateCompound(0)` — cluster-level Pareto points for network
28 $\mathcal{C} \leftarrow \{\bar{c} \cdot G_{\mathrm{ct}}(\bar{c}) \mid \bar{c} \in \mathcal{C}_{\mathrm{clus}}\}$ — derive task quality metrics
29 $\mathcal{C} \leftarrow \mathcal{C} \downarrow I_{\mathrm{Mlow}}$ — abstract from lower-level metrics
30 $\mathcal{C}_{\mathrm{opt}} \leftarrow \min(\mathcal{C})$ — task-level Pareto points

Algorithm 4.5: Incremental minimisation function

```
function AddAndMin(C, c̄):
    for all ā ∈ C:          loop through all configurations ā in C
        if ā ⪯ c̄:           if c̄ is dominated by ā, do not add c̄ and return
            return C
        else if c̄ ⪯ ā:      if c̄ dominates ā, remove ā
            C ← C\{ā}
    return C ∪ {c̄}          c̄ is not dominated by any configuration in C, so add it
```

4.2.1 Interleaved Combining, Deriving and Minimising

Since a product set can be very large, an implementation that first computes the whole product set and then derives the metrics and minimises the resulting set would need an excessive amount of memory. Therefore, the loop body of Algorithm 4.3 is rewritten in an interleaved fashion: when an element of the product set is computed, metrics are immediately derived, and the resulting configuration is then integrated in a configuration set that is kept minimal, as in the following construct.

```
C_min ← ∅                          initialise minimal set
for all c̄ in Product(S):           iterate through product set
    c̄ ← c̄ · F(c̄)                   append derived metrics
    C_min ← AddAndMin(C_min, c̄)    add to set and keep Pareto minimal
```

The `Product` function yields a new element from the free product of the configuration sets in the list `S`, instead of computing the whole product set at once. Thus, `Product` is a *generator* as available in programming languages such as Python and C#. The incremental minimisation function `AddAndMin`, given in Algorithm 4.5, is derived from the Simple Cull algorithm [22]. This function adds a new configuration to a Pareto-minimal configuration set, and keeps the set minimal. This construct is integrated in Algorithm 4.4. The asymptotic time complexity of this approach is the same as before, but the memory demand is now in the order of the size of the minimised configuration sets, instead of the product sets, and thus a lot smaller.

4.2.2 Quantisation

The metric values that are obtained from the mapping functions are of limited accuracy, and we can take this into account while performing minimisation. Given a configuration set with two example configurations $\bar{a} = (20, 0.1)$ and $\bar{b} = (2, 0.1000001)$, we see that $\bar{a} \not\preceq \bar{b}$ and $\bar{b} \not\preceq \bar{a}$, and hence both are Pareto optimal. However, we may consider the difference between the values

in the second quantity to be insignificant. Looking at only the first quantity, $\bar{a}$ is clearly better (assuming larger is better). We decide to treat both second-quantity values as being equal to 0.1, such that $\bar{a} \preceq \bar{b}$, and $\bar{b}$ will be removed by minimisation.

We capture the accuracy for each quantity in a vector $\bar{q}$, and define a function $R_{\bar{q}} : \mathcal{S} \to \mathcal{S}$ that quantises a configuration. $R_{\bar{q}}$ rounds every value in a configuration to the nearest multiple of the corresponding value in $\bar{q}$. This kind of quantisation is trivially monotone (rounding does not change the order of configurations in a quantity), which ensures that quantised Pareto points correspond to Pareto points in the unquantised set. The difference is that multiple (Pareto) points may have equal metric vectors after quantisation, and only one of those coinciding configurations is kept. We use this function in line 20 of Algorithm 4.4 to extend the configuration with quantised metrics, while the original metrics are hidden for the minimisation operation in the next line. In line 23, we unhide the original metrics and abstract from the quantised ones, because we use the unquantised configurations for further processing to prevent the propagation of rounding errors.

4.2.3 Indexing

Recall that a cluster configuration is a vector of parameter values (for all nodes in the cluster) plus a vector of metric values. A straightforward implementation of the algorithm would store each configuration exactly in this format. However, as illustrated in Figure 4.3, the metrics of a cluster can be directly computed from the metrics of lower-level clusters by incremental mapping functions, for the networks we study; the parameters are not needed. Algorithm 4.3 does use incremental mapping, but still leaves the full parameter vectors (hidden) in the configuration sets. Our next optimisation measure uses this opportunity to save on precious storage space and time-consuming memory accesses.

First of all, in the case that multiple nodes have the same parameter spaces, we need to store this space only once, and in the configurations we can use indices to parameter vectors, instead of the full vectors. Still, a cluster configuration would contain a parameter index for each node contained in the cluster, and therefore the size of configurations grows when the cluster algorithm progresses and clusters become larger. As a result, the memory demand of the algorithm does not scale, which may be prohibitive for large networks. We therefore extend the use of indexing in the cluster algorithm, as shown in Figure 4.5. In each step of the algorithm, a (one-node) root cluster is combined with its child clusters. For each of these clusters we need a configuration set with metric vectors. Each metric vector in a configuration set is given an index, and when combining metric

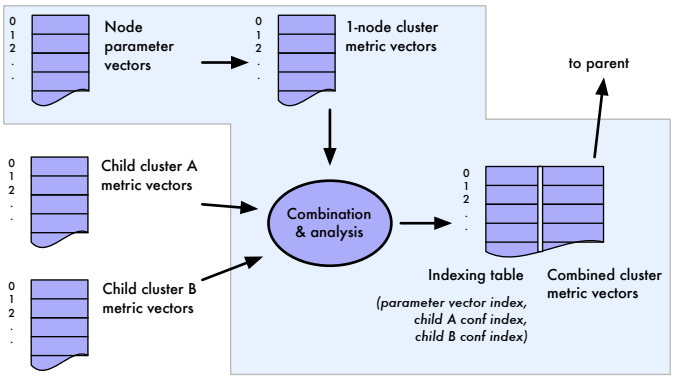


Figure 4.5: Implementation of one clustering step (node degree 2). The figure shows the tables that are needed for configuration sets and indexing, which constitute almost all of the memory needed. The numbers shown to the left of the tables are the indices.

Algorithm 4.6: Reconstructing a parameter vector. `Index[`n`][`i`]` *selects the* i^{th} *index in the indexing table of node* n*. The first column in an indexing table is always the parameter index for the node; the remaining columns are indices of configurations of child clusters.*

```
1  function reconstructParams(n, i):
2      k̄ ← Index[n][i]                    get the i^th row in the indexing table of node n
3      p̄ ← S_{Pc,n}|_T[k̄[0]]              obtain the parameter vector for node n
4      if n is a leaf node:
5          return p̄                       return parameter vector for leaf node n
6      j ← 1
7      for each child c of n:             recursively reconstruct descendents' parameters
8          p̄ ← p̄ · reconstructParams(c, k̄[j])
9          j ← j + 1
10     return p̄                           return parameter vector for cluster with root n
```

vectors into new configurations, their indices are stored with them. After deriving quality metrics and minimising the new configuration set, this set is split into a set that holds only the metric vectors and a linked set that contains vectors of indices, called the *indexing table*. Subsequently, only the table of metric vectors is used in the next cluster step.

Algorithm 4.4 implements this approach. When creating one-node clusters in the `CreateOneNode` function, a hidden index is prepended to each configuration in line 3, and after deriving quality metrics, the parameter values are abstracted from in line 4. Also the compound clusters created by `CreateCompound` are prefixed by an index in line 24. The configurations in the product set now contain, instead of the parameter vectors for all contained nodes, the indices of the configurations of the contained clusters. After the product/derive/minimise loop, the indexing table is extracted in line 22, and the set of metric vectors is returned with new indices prepended (line 23–25). The indexing tables are stored in a data structure called `Index`, to be used later.

Thus, besides the parameter spaces $\mathcal{S}_{\mathrm{Pc},i}|_T$, for every node in the network only an indexing table is stored. Memory for intermediate cluster configuration sets can be freed immediately after usage in the next cluster step. When the final set of Pareto points (for the whole network) has been computed, one metric vector will be selected, and we need to reconstruct the parameter vector that gives rise to it by tracing back through the indexing tables. Algorithm 4.6 shows how this is done by a recursive function that is called with the ID of the network's root and the index of the selected configuration.

4.3 Distributed Execution

Algorithm 4.4 is given as a centralised algorithm that is run separately from the WSN, before starting the WSN's task. However, the algorithm treats nodes in a leaf-to-root fashion: a node only depends on information from descendants to compute configurations for the whole cluster with itself as root. It is therefore also possible to execute the algorithm in a distributed way, in which each node passes the optimal configurations on to its parent, after computing the Pareto set for its cluster. When the network's root has been reached, the Pareto optimal configurations for the whole network are known.

In contrast to the centralised algorithm, distributed QoS optimisation requires communication in the network. The use of indexing in Section 4.2.3 limits the communication overhead per node to just the transmission of the metric vectors of its Pareto-optimal (cluster) configurations. Furthermore, the communication costs of loading the chosen configuration to the network (phase 5) are also reduced. In the centralised case, the nodes need to transfer a packet containing the selected parameter vectors for each of its descendants. In the distributed case, however, only one index per *immediate* child node needs to be transmitted. Nodes use their indexing tables to find out which parameter vector they need to use, and which indices need to be sent to their children. All transfers in the QoS optimisation and downloading phases should be reliable, and therefore an automatic repeat request scheme (acknowledgements and retransmissions) is used.

Since each node computes its own node-level configurations in a distributed QoS-analysis algorithm, it is not needed to gather the local details of nodes (node type, parameter set, energy level, etc.) at a central point. The initialisation phase of configuration therefore becomes simpler. In fact, if also the tree-construction phase is done in a distributed way, there is no initialisation needed altogether. Figure 4.6 shows a state diagram featuring the *QoS optimisation* and *loading* states, which correspond to the equally named phases of the configuration process. This diagram

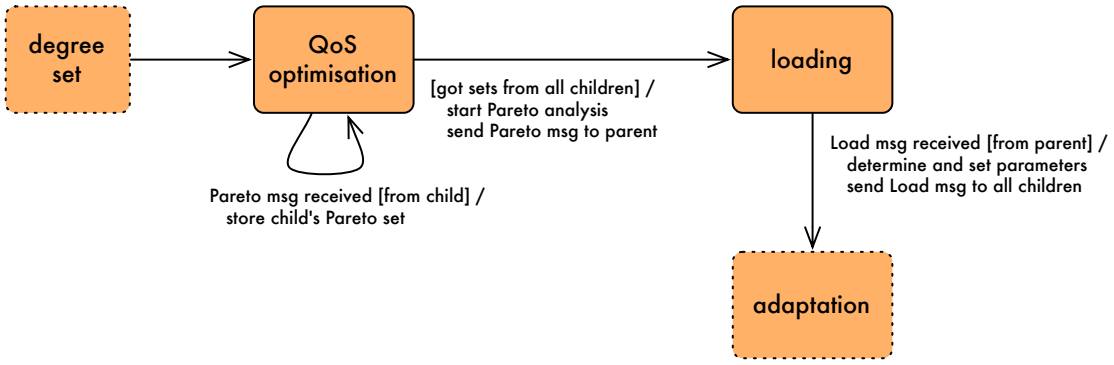


Figure 4.6: Distributed QoS optimisation, state diagram. Assumed is that a distributed tree-construction algorithm is executed first that has a final state degree set*, after which a node knows its parent and children (see Chapter 5). State transitions are triggered by events and/or conditions as annotated at the arrow before the slash. Actions at a transition are given after the slash. All events that are not listed at a state are ignored.*

applies to all nodes except the root. The preceding distributed tree algorithm is described in detail in Chapter 5. After this algorithm, the node knows its parent and children. The action *start Pareto analysis* in the diagram is implemented as Algorithm 4.4 without the recursion, in which the list of child clusters Pareto sets S is constructed using the Pareto messages received from the children. If the node has no children (a leaf node), it immediately starts Pareto analysis. The resulting Pareto set is sent in a Pareto message to the parent. Subsequently, the node waits in the loading phase for a Load message from its parent, containing the index of the selected configuration. Upon receiving this message, the node configures itself, and sends Load messages to its children. Next, the task can be started, and the run-time adaptation state is entered, which is described in Chapter 6.

The root node, when in possession of the Pareto set of the whole network, enters the *selection* state in which it chooses one of the configurations, instead of transmitting a Pareto message. After that, it initiates the loading phase by sending a Load message to each of its children.

As the distributed execution allows for computations to run in parallel (all leaf nodes can start at the same time), there is a scalability benefit. The total run time for the centralised (T_{centr}) and distributed execution (T_{distr}), when ignoring communication delays (we learn from the experiments in Section 4.6 that communication time is negligible), can be expressed as follows:

$$T_{\text{centr}} = \sum_{i \in V} T(i), \tag{4.5}$$

$$T_{\text{distr}} = \max_{i \in V} \sum_{j \text{ on } \bar{p}^i} T(j), \tag{4.6}$$

where $T(i)$ is the run time of the cluster step with node i as root, and $\vec{p}^{\,i}$ is the path from node i to the network's root node. Thus, the run time of the distributed execution is determined by a critical path of node run times, instead of the run times of all steps together. On the other hand, sensor nodes usually have very simple processors, while the centralised algorithm can be executed on a fast and powerful server. Moreover, running on sensor nodes uses their batteries, a scarce resource.

The pros and cons of a distributed QoS-optimisation algorithm in the configuration process should be weighed carefully, considering the situation at hand. An important observation is that the in-network computation of configurations is a prerequisite for the development of *localised* reconfiguration methods, to quickly respond to changes in the local environment, which is the topic of Chapter 6. Another interesting option would be a hybrid system, in which a number of more powerful nodes are deployed as cluster heads. These special nodes could take care of the configuration of the nodes under their supervision in a centralised way, in order to save resources in sensor nodes. This mix of the centralised and distributed approaches may provide an interesting trade-off. Section 4.6.4 shows results for distributed QoS optimisation, which are based on a TinyOS implementation for TelosB sensor nodes.

4.4 Complexity Control

From Section 4.1.5 we learn that a major factor that determines the run time of a cluster step – which may take place at a single sensor node in the distributed approach – is the size of the product set $|\mathcal{C}_{\text{prod}}|$: the product of the sizes of the cluster configuration sets that are being combined. The complexity further depends on the number of Pareto points of the compound cluster ($|\mathcal{C}_{\text{min}}|$). Both sizes are not predictable in general, but we can still do a number of things to influence them. Measures to decrease the size of the product set could focus on either reduction of the *number* of clusters that are combined (equal to the node degree plus one), or the *size* of the child cluster's configuration sets. The former is dealt with in Chapter 5 about routing tree construction. In this section, we discuss ways to control the complexity of the cluster algorithm by limiting the sizes of the sets contributing to $\mathcal{C}_{\text{prod}}$, in exchange for lower-quality results (the meta trade-off between configuration cost and task quality).

Besides reducing the configuration time, there is another reason for placing such limits, if the QoS-optimisation algorithm is executed in a distributed way on the sensor nodes themselves, as outlined in Section 4.3. These sensor nodes usually have a very limited amount of memory

available, and therefore it may be needed to limit the size of configuration sets in the algorithm, if there is simply no space to store larger sets. Moreover, for a WSN operating system such as TinyOS, the memory allocation is static, and suitable fixed sizes of all data structures need to be determined in advance.

4.4.1 Limiting the Complexity of the Cluster Algorithm

The run time of a cluster step in Algorithm 4.4 is the time needed for one call to the function `CreateCompound` without counting the time needed for recursive calls in this function in line 15. Looking closely at the code of the algorithm, we see the following structure, where $p = |\mathcal{S}_{\mathrm{Pc},i}|$, $q = |\mathcal{C}_{\mathrm{prod}}|$, $r = |\mathcal{C}_{\mathrm{min}}|$:

```
1 function CreateCompound(i):                    O(p + q^2 + r)
2     C ← CreateOneNode(i)                       O(p)
3     ...                                        O(1)
4     for all c̄ in Product(S):                   O(q^2)
5         ...                                    O(1)
6         C_min ← AddAndMin(C_min , c̄)           O(r)
7     ...                                        O(r)
```

We assume p is a relatively small, known and fixed number, so we ignore it and focus on q and r. Both q and r are unpredictable, and depend on the model and values of the parameters throughout the network. The worst-case complexity of the `AddAndMin` function is $\mathcal{O}(r)$: if $\bar{c}$ is a Pareto point, it is compared with all configurations in $\mathcal{C}_{\mathrm{min}}$ and then added. The worst-case for the loop is when every configuration in the product set is a Pareto point, so r grows in each iteration. Hence, the loop is $\mathcal{O}(q^2)$, and (as $r \leq q$) the overall complexity of a step is also $\mathcal{O}(q^2)$. Moreover, as explained in Section 4.1.5, the complexity of the full algorithm is exponential in the number of nodes in the worst case, as q may grow exponentially with each step. However, if we could limit q to a constant Q, the complexity of the cluster step becomes a constant $\mathcal{O}(Q^2)$. If we do this for each step, the overall complexity of Algorithm 4.4 becomes $\mathcal{O}(|\mathcal{N}| \cdot Q^2)$, and hence it is guaranteed to be linear in the number of nodes. If we also restrict r to a constant $R < Q$, the complexity becomes $\mathcal{O}(|\mathcal{N}| \cdot Q \cdot R)$. As said above, setting the limits Q and R is needed for an implementation on memory-constrained sensor nodes, and hence very relevant in practise.

Restricting the size of $\mathcal{C}_{\mathrm{min}}$ or $\mathcal{C}_{\mathrm{prod}}$ implies a potential reduction in quality, as a number of Pareto points may need to be left out. The magnitude of the quality loss depends on the number of Pareto points, which is not predictable, and on the choice of points that are kept. It is hard to steer the choice of points in case of a bound on $\mathcal{C}_{\mathrm{min}}$, as it depends on the order in which the points

in the product set are generated in the loop. For a reduction of $\mathcal{C}_{\text{prod}}$ we have more freedom. This could be done by assessing the size of the product set before combining, and reducing child-cluster configuration sets if the product set is larger than the threshold Q. The key questions are then: *which sets* to reduce, *how many configurations* to remove, and *which configurations* to remove from a set. Here we need to assess the impact (loss of quality) of removing a configuration. Our approach to address the first two questions is to reduce the size of the largest configuration sets first, until the product size is smaller than the threshold. By doing this, the sets will become of similar size, such that every combined cluster maintains a diverse set of solutions (good distribution and spread, see below). For example, if we have three sets of sizes 10, 8 and 5 configurations (a product set of 400 configurations), and a product threshold of 200, we will reduce the sets to 6, 6, and 5 configurations respectively (a product of 180).

4.4.2 Reducing Pareto Sets

Suppose a given Pareto set $\mathcal{C}$ needs to be capped at a maximum of m points. Which points to remove? The easiest way is to just randomly remove configurations. However, for a configuration set, it seems to be desirable to have a good *distribution* and *spread* of points. A good spread means that the configuration set spans the whole (reachable part of the) metric space. Distribution refers to the placement of points in the metric space; the points should be as far away from each other as possible to have an even distribution across the space. We use this as a heuristic when reducing sets.

The goal is to form an m-point subset $\mathcal{C}_m$ of an n-point Pareto set $\mathcal{C}$ with the best spread and distribution possible. A number of approaches are described in the literature [42, 57, 73], which are based on data clustering: based on some criteria, m clusters of points are formed and per cluster, one representative point is chosen. Since speed is of utmost importance for an implementation on sensor nodes, we use a simpler method with a random component (see below). We first define the problem as follows.

Definition 4.4 (Distribution Factor).

$$\eta(\mathcal{C}_m) = \min_{\bar{c}_0, \bar{c}_1 \in \mathcal{C}_m} d(\bar{c}_0, \bar{c}_1), \tag{4.7}$$

where $d(\bar{c}_0, \bar{c}_1)$ is the distance between configurations $\bar{c}_0$ and $\bar{c}_1$, for which we use the average

Algorithm 4.7: Computing a well-distributed k-point subset of $\mathcal{C}$

```
function Reduce(C, k):
    allocate 3-column table
    for each c̄0, c̄1 ∈ C:
        add row d(c̄0, c̄1), c̄0, c̄1 to the table
    sort rows of table by ascending distance
    while |C| > k:
        r ← first row of the table
        randomly choose a c̄ from the configuration pair in r
        remove all rows in the table containing c̄
        C ← C\{c̄}
    return C
```

of the normalised differences per quantity:

$$d(\bar{c}_0, \bar{c}_1) = \frac{1}{k} \sum_{i=0}^{k-1} \frac{|\bar{c}_0(Q_i) - \bar{c}_1(Q_i)|}{Q_{i,\max}}, \tag{4.8}$$

with k the number of quantities, $\bar{c}(Q_i)$ the value of configuration $\bar{c}$ for quantitiy Q_i, and $Q_{i,\max}$ the largest value of quantity Q_i over all configurations in $\mathcal{C}$.

We are looking for the m-point subset $\mathcal{C}_m$ that has the largest distribution factor and use this as our reduced configuration set. This set has the best distribution and typically also the best spread (intuitively, the best distribution can be achieved if the whole space is used). Note the differences with Definition 2.3: $\mathcal{C}_m$ is *not* defined as the m-point subset with the smallest quality loss compared to $\mathcal{C}$. We are only interested in quality loss at the network level; reduction is typically used at lower levels, and there is no clear relation between the quality loss at various levels. We therefore choose to apply the commonly used distribution-based way of reduction.

To find $\mathcal{C}_m$, we could simply try all m-point combinations of points from $\mathcal{C}$ and determine $\eta(\mathcal{C}_m)$. This approach takes $\mathcal{O}(\binom{m}{2} \cdot \binom{n}{m})$. More efficient exact algorithms might be possible, but this is left for future work. For our purpose it seems to be sufficient to use an approximated, but much faster way of computing $\mathcal{C}_m$. We suggest Algorithm 4.7, which runs in $\mathcal{O}(n^2)$. It first determines the distances between all pairs of points in $\mathcal{C}$ according to distance metric d in Definition 4.4. It then repeatedly removes one of the points (randomly chosen) in the pair with the shortest distance until m points remain. See Figure 4.7 for an example. Section 4.6.5 contains results on the effects of limiting the size of the product set.

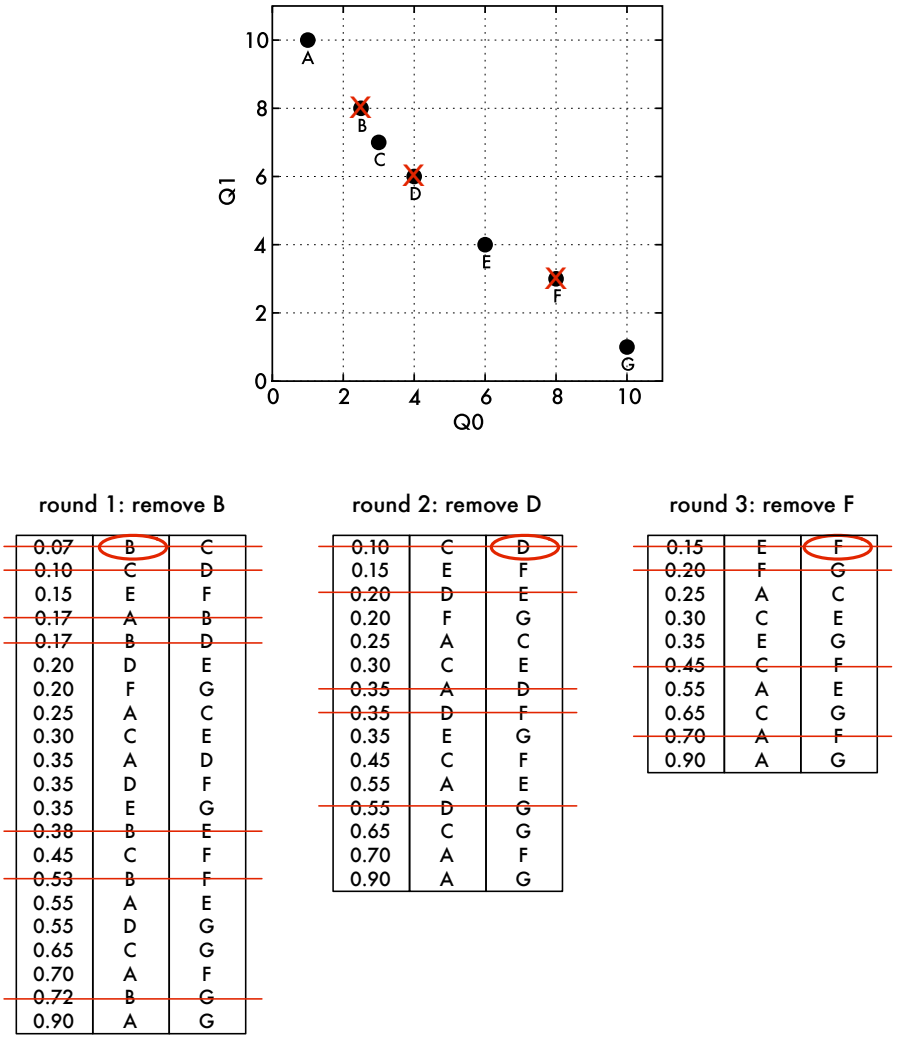


Figure 4.7: The Pareto set above contains seven configurations, of which we want to keep only four. Algorithm 4.7 is a heuristic algorithm that attempts to remove points, such that the remaining configuration set has a good spread and distribution. The algorithm starts by building a table that contains all pairs of points with the distances according to (4.8), and sorting this table by ascending distance (left-most table). Subsequently, configurations are removed in rounds, until the desired number remains. In each round, one of the points in the pair with the shortest distance is chosen (randomly) and removed. Then all table entries containing the removed configuration are erased as well. In the example above, configurations, B, D, and F are consecutively removed. The resulting distribution factor η is equal to 0.25.

4.5 Multiple Tasks

The problem description in Chapter 3 targets the configuration of a WSN for a single task. It is already possible with the current method to analyse heterogeneous networks, in which any node may be different. Each (type of) node needs its own parameter space $\mathcal{S}_{\mathrm{Pc},i}$ and model to map parameters to cluster metrics. At the cluster and task levels, however, all metrics need to be the same, which makes sense if all nodes are working together to perform a single task.

In some cases it is even possible to configure a network for multiple tasks running at the same time, using the same method. For this to work, it is needed that both tasks share a common routing tree, and that the cluster/task metrics of all tasks are merged into a single metric space. We show an example of this below.

While using a shared configuration space for multiple tasks is useful for relatively static applications, an interesting next step would be to allow any combination of tasks to share a WSN. Ideally, we would analyse each task separately, and then select a feasible Pareto point for each task, taking into account the sharing of resources. This is a challenging topic that we do not cover in depth in this thesis, but leave for future work. We do sketch a possible solution in this section.

4.5.1 Shared Configuration Space

As an example, we study a network running both SM and TT at the same time. We have three node types: nodes that do either SM or TT, and nodes that do both. The network can contain any distribution and any number of these node types. One practical scenario to use this could be a disaster scene in which we constantly need to observe the temperature across the region to be aware of fire, while at the same time we want to track people walking around. We may specify that the tracking information needs to be more fine-grained than the temperature mapping task, and hence we could deploy a large number of TT nodes plus a smaller number of SM+TT nodes (assuming that using combined nodes is more cost effective than using separate SM nodes).

We let both tasks share the same routing tree, which means that TT nodes also relay traffic for the SM task and vice versa. We assume that SM and TT need different sensors, so the combined SM+TT nodes need to have two sensors, and hence two sample-rate parameters. It is therefore needed to create a new node-level model (mapping functions) for the combined SM+TT nodes. The quality metrics in this model are the union of the metrics of the SM and TT models. Further, a new cluster model is needed. Note that for the cluster method to work, all clusters need to have the same metrics. And since a cluster can now have both the SM and TT tasks, we need

Table 4.2: Metrics for combined SM/TT clusters

Metric	Task
Information Completeness	SM
Information Completeness	TT
Reporting Rate	SM
Detection Speed	TT
Lifetime	SM/TT
Coverage Degree	SM
Coverage Degree	TT
Output Traffic (additional metric)	SM/TT

the quality metrics for both tasks incorporated in the configuration space; see Table 4.2 for an overview of the metrics. Note that some metrics are shared by both tasks, while others are not. For example, we are interested in the individual coverage of the tasks, but we define a shared lifetime metric because both tasks share the same network.

The mapping functions can be easily derived from the functions in Table 4.1. However, they do depend on the type of the root node of the cluster. For instance, if the root is an SM node, it only forwards TT traffic and it does not add a new detection-delay term. Therefore, (4.4c) will just be

$$S^{\mathrm{c}}(c) = \min_{i \in ch(c)} \left\{ \left(\frac{1}{S^{\mathrm{c}}(i)} + D_{\mathrm{tx}} \right)^{-1} \right\}.$$

There are similar considerations for the other metrics. Most importantly, all mapping functions are still monotone, so the cluster algorithm will return all Pareto-optimal configurations.

4.5.2 Decoupled Task Optimisation

It is interesting to consider a WSN as a platform on which multiple tasks can run simultaneously, while sharing the platform's resources, especially if one would allow each task to have its own sink node and routing tree. All tasks may be known in advance (at configuration time), or could be started and terminated by the user while the network is operating and running other tasks. In such a scenario, it is convenient to be able to analyse tasks separately from one another, and then combine the per-task results. Phases 1 to 3 of our current analysis method deliver a complete set of feasible Pareto-optimal configurations for a single task, representing all potentially suitable trade-offs. In phase 4, the selection phase, a choice is made for one of these points. If we have the Pareto sets for all tasks that should run on the WSN, we may turn the selection phase into a multi-task selection phase, in which a feasible configuration for each tasks needs to be selected.

Since the platform resources are shared by all tasks, the resource metrics and constraints play a major role in this selection phase.

The problem sketched here resembles a Multi-Dimensional Multiple-Choice Knapsack Problem (MMKP) [1, 29, 70], a variant of the 0–1 Knapsack Problem: *multiple choice* means that each task has a choice of multiple Pareto points, and *multi dimensional* refers to the multiple resources and constraints. A MMKP is generally specified as follows.

Maximise

$$Z = \sum_{i=1}^{m} \sum_{j=1}^{n_i} p_{ij} x_{ij}, \tag{4.9}$$

subject to

$$\sum_{i=1}^{m} \sum_{j=1}^{n_i} w_{ijk} x_{ij} \leq c_k, \qquad k \in \{1, \ldots, l\}, \tag{4.10}$$

$$\sum_{j=1}^{n_i} x_{ij} = 1, \qquad i \in \{1, \ldots, m\}, \tag{4.11}$$

$$x_{ij} \in \{0, 1\}, \qquad i \in \{1, \ldots, m\}, \quad j \in \{1, \ldots, n_i\}. \tag{4.12}$$

In this formulation, there are m item classes (task configuration sets), each having n_i items (configurations), and l resources. Each item j of class i has a non-negative value p_{ij} (a quality-metric vector should be mapped to a single value), and requires resources $w_{ij} = (w_{ij1}, w_{ij2}, \ldots, w_{ijl})$. The resource constraints are captured in a vector $c = (c_1, c_2, \ldots, c_l)$ of upper bounds. A variable x_{ij} can be either 0 or 1, reflecting whether the corresponding item is picked or not. Values and resources are additive. Exactly one item from each class is selected to maximise the total value, subject to the resource constraints. The MMKP is NP-hard.

The problem is especially challenging as resources appear at the node level, and their number therefore depends on the number of nodes. Also, communication bandwidth is a shared resource in some sense (for neighbouring nodes) and a distributed resource in some other sense (nodes far apart); how to model this? Furthermore, resource metrics should no longer be hidden or abstracted from after matching with the constraints, such that the Pareto sets contain trade-offs between quality and resource metrics, instead of between quality metrics alone, which potentially leads to a very large number of Pareto points.

The MMKP formulation treats all parameters – or the configurations of various tasks – as

independent. In the WSN case, parameters may be shared between multiple tasks, and should therefore match in the configurations of different tasks. This consistency constraint potentially alleviated the complexity of the MMKP, as it reduces the configuration space.

A potential solution to the MMKP is very similar to the cluster algorithm. The m items in the problem description may be combined incrementally, while applying constraints and minimising in each step. The incremental mapping functions (additions) are trivial and monotone. Also the reduction technique from Section 4.4 can be exploited to trade quality for complexity. The consistency constraint (WSN tasks are not independent) can be incorporated by the join operator of Pareto algebra [23], which combines a free-product and a constraint to match quantities in two configuration sets (similar to the join function in relational databases). Such an approach, in the domain of chip-multiprocessors, has been published by Shojaei et al. [60]. Further exploring this approach, or deriving useful heuristics, is an interesting direction for future work.

4.6 Experiments

We implemented Algorithm 4.4 and ran it for networks of different sizes, for both the Spatial Mapping and Target Tracking models as given in Section 3.2. The computations intended for centralised processing were implemented in C++ (with our Pareto-algebra library [22]) and carried out on a laptop with Intel Core 2 Duo processor (using only one core) at 2.4 GHz and 2 GB RAM. These results can easily be scaled to other platforms. To assess the performance of the distributed algorithms on real sensor nodes, we gathered profiling data from an implementation in TinyOS [36] on a TelosB sensor node [16] (see Section 4.6.4), and used this information in simulations of a whole WSN in the OMNeT++ simulator [64]. The simulations allow the loss of packets with a probability depending on the distance between sender and receiver, and due to collisions. We used CSMA-based medium-access control, and Automatic Repeat Request (ARQ) was implemented where needed to achieve reliable communication.

For each network size, we randomly distributed sensor nodes in a square area. To ensure an even distribution across the area, we placed the nodes with a certain variance around fixed grid points. While scaling the number of nodes, the area was scaled accordingly, such that the node density was equal for all networks. For each network, the transmission range was set to 20 m, and a routing tree was created. To ensure a fair comparison between the results for all networks – algorithm complexity depends on node degree – we only used SPSTs in which each node has at most three immediate child nodes. See Chapter 5 for details on how to construct such trees.

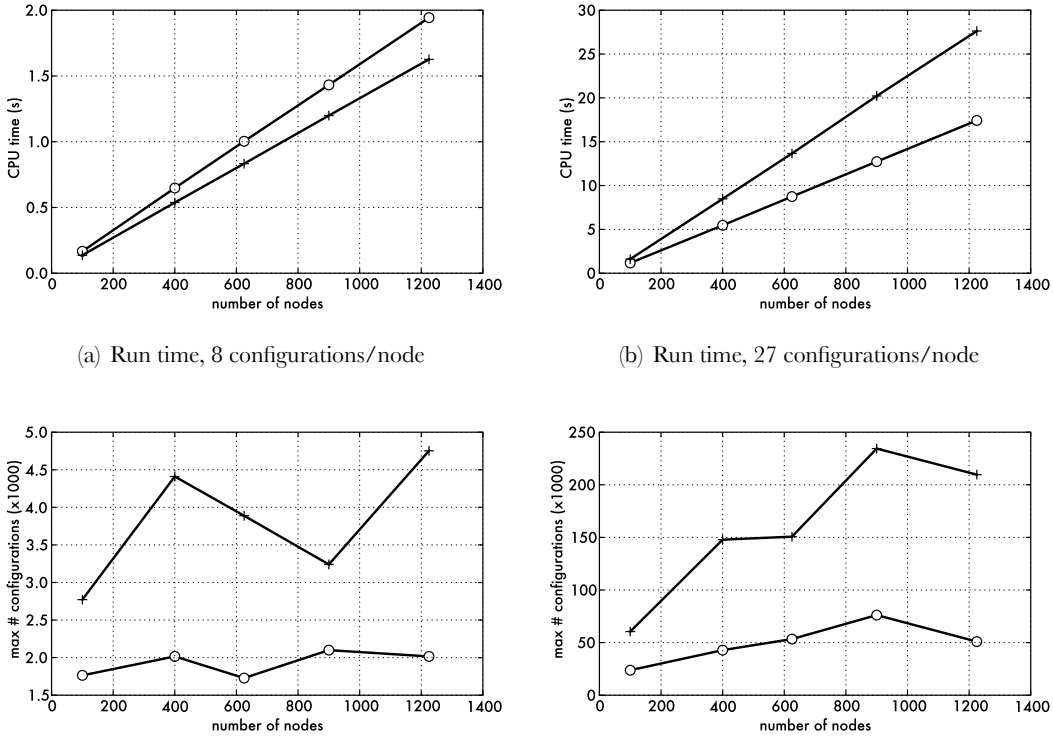
We first set the quantities for the node-level parameters as follows: $TxPower = \{0, -5, -10\}$ (dBm), $SampleRate = \{0.5, 0.3, 0.1\}$ (Hz), $DutyCycle = \{0.2, 0.4, 0.6\}$. This leads to $3^3 = 27$ possible configurations per node. Subsequently, we did the same tests for just 8 configurations per node, by omitting the last parameter value in each quantity. To achieve some robustness in the measured run times and configuration counts, for each network size and number of configurations, we analysed 100 different networks.

4.6.1 Run Time and Number of Configurations

To gain insight in the scalability of Algorithm 4.4, we recorded the size of the product set ($|\mathcal{C}_{\text{prod}}|$) in each step, as well as the run time, based on a centralised (sequential) execution. Constraints were not used in these experiments; the goal was to find *all* Pareto points. It turned out that the maximum size of $\mathcal{C}_{\text{prod}}$ (over all steps), the number of configurations simultaneously considered, stays limited, even when the network size increases. For the tests with 8 configurations per node, this maximum was $4,752$ (see Figure 4.8). For the case of 27 configurations per node, $|\mathcal{C}_{\text{prod}}|$ increased to about $234.4 \cdot 10^3$. Moreover, judging from Figure 4.8, the average run time of the algorithm increases roughly proportionally with the number of nodes in all scenarios, which is good considering that the underlying configuration space grows exponentially. Therefore, we may conclude that the algorithm is very well scalable and thus suitable for the configuration of a WSN. For example, for the 900-node networks, the TT scenario, and 27 configurations per node, the algorithm took on average 20.2 seconds to complete, while the total number of possible configurations is 27^{900}. The resulting Pareto set for one of these networks is given in Table 4.3. There are 9 Pareto-optimal configurations, and each of these solutions has a corresponding set of parameter values for each node. We see that there are clear trade-offs between most quality metrics.

4.6.2 Memory Usage

Further, we recorded the average memory usage (over all steps) of the cluster algorithm, in the same experiments as above. Figure 4.9 shows the difference in memory usage between Algorithm 4.3 and Algorithm 4.4. It is clear that the average memory usage of the optimised implementation is nearly independent of the network size, while the non-optimised implementation needs more memory for larger networks.



(a) Run time, 8 configurations/node

(b) Run time, 27 configurations/node

(c) Max. size $\mathcal{C}_{\mathrm{prod}}$, 8 configurations/node

(d) Max. size $\mathcal{C}_{\mathrm{prod}}$, 27 configurations/node

Figure 4.8: Run time and number of configurations. For increasing network size, the run time (a and b) does not grow more than linearly for both the SM (circle markers) and TT (cross markers) scenarios and two different sizes of the parameter space. The reason for this linear growth is that the maximum number of simultaneously considered configurations at any clustering step ($|\mathcal{C}_{\mathrm{prod}}|$) stays limited (c and d).

Table 4.3: Analysis results (Pareto points) for 900-node example network.

Information Completeness I^{c} (%)	Detection Speed S^{c} ($\frac{1}{\mathrm{s}} \cdot 10^3$)	Lifetime T^{c} (h)	Coverage Degree C^{c}
84	41	3481	0.2
63	41	3773	0.2
2.0	41	4073	0.2
78	41	1821	0.4
59	41	1970	0.4
2.0	41	2123	0.4
78	41	1214	0.6
59	41	1313	0.6
2.0	41	1415	0.6

Algorithm 4.8: Genetic algorithm (SPEA)

```
1 generate initial population P and empty P'
2 repeat:
3     copy non-dominated points from P to P'
4     minimise P'
5     prune P' to a maximum size, if needed
6     calculate the fitness of individuals in P and P'
7     select individuals from P ∪ P' up to the population size
8     apply crossover and mutation
9     break if stop criterion is satisfied
```

Table 4.4: Settings used for the genetic algorithm

Setting	Value
Population size	200
Maximum size of P'	20
Crossover probability	0.5
Mutation probability	0.2

4.6.3 Comparison with a Genetic Algorithm

We also explored the configuration space of the example network of Table 4.3 via the genetic algorithm SPEA [73]. A WSN configuration is represented by an individual in the genetic algorithm, that has one chromosome (the parameter vector) made up of genes (parameter values), and a fenotype (the metric vector). SPEA uses two sets of individuals (configuration sets): the population P and the non-dominated set P'. Algorithm 4.8 shows a high-level overview of the algorithm. We use a random initial population P in line 1. Lines 3 and 4 take the Pareto points from the previous iteration as the set P'. The prune statement in line 5 is similar our reduction algorithm in Section 4.4: it reduces a Pareto set to a maximum size by removing points. The fitness function of SPEA in line 6 assigns a strength (or fitness) to each individual, based on the number of individuals it dominates (for P'), or the strengths of dominating individuals (for P). Line 7 selects points from P and P' by a game based on the strength values. Finally, crossover (mixing two chromosomes into a new one) and mutation (changing some genes randomly) is applied to the remaining individuals. The algorithm's main loop is repeated until the stop criterion in line 9 is satisfied. The settings that we used in our experiments are summarised in Table 4.4. The crossover probability of 0.5 means that two chromosomes are equally mixed into a new one (the genes from either chromosome are equally likely to appear in the new chromosome).

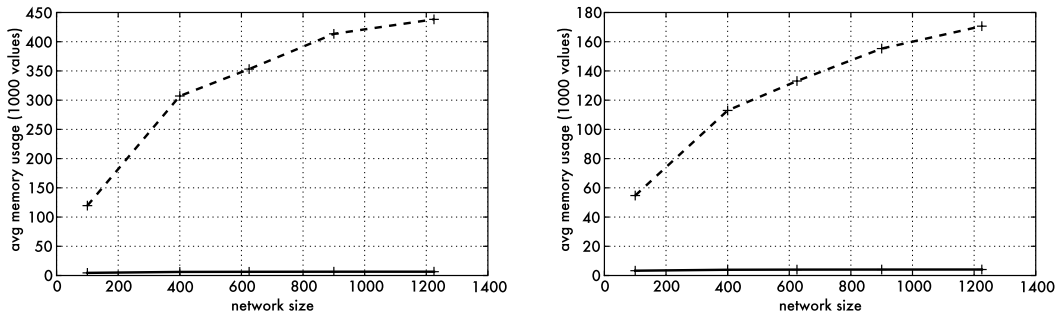
We ran this algorithm on the example network without a stop criterion. Even after running

for three days, it returned 6 configurations (among others $(0.8, 12, 3869, 0.2)$), which were all strictly dominated by at least one configuration in Table 4.3. It turns out that configurations with the best metric values are so rare and isolated in the total space of size 27^{900}, that the genetic algorithm is doomed to fail: the probability of finding the Pareto points goes to zero. The best metric values found were 8.4, 12, 3869 and 0.2 (order as in the table), which is 90%, 70%, 5% and 67% lower than the best values found by our method. This result confirms the expected result that a search space of 27^{900} is too large to search efficiently and accurately via a randomised approach, and it emphasises the strength of our exact algebraic approach.

4.6.4 Distributed QoS Optimisation

The experimental results above are all for a centralised execution of Algorithm 4.4. However, Section 4.3 shows that the algorithm can also be run directly on the sensor network in a distributed fashion. For these experiments, we use profiling information from an implementation in TinyOS for the target-tracking task. We executed this program on a TelosB sensor node, and measured the run time of one cluster step for various sizes of the product set ($|\mathcal{C}_{\text{prod}}|$). The run time is to a large extent determined by $|\mathcal{C}_{\text{prod}}|$, since it represents the number of configurations that need to be analysed. Figure 4.10 shows the results: it appears that the run time has an approximately linear relation with $|\mathcal{C}_{\text{prod}}|$, with a slope of 0.0234 s per configuration in $\mathcal{C}_{\text{prod}}$. TelosB nodes have very basic processing capabilities: they have a TI MSP430 microcontroller (a 16-bit RISC processor) that runs at 8 MHz, has 10 kB of the data memory (RAM), 48 kB program flash memory, and a transceiver with a bitrate of 250 kbps for communication. Our TinyOS implementation uses about 18.3 kB of the program memory and 4.6 kB RAM when targeted at 27 configurations per node, a node degree of at most three, and child configuration sets of size eight.

We used simulations to find the run time of the cluster-based QoS-optimisation algorithm on a whole WSN, including communication overhead, based on this profiling information for TelosB nodes. The simulated nodes run the same program as the real nodes, and the simulated processing time is taken as $0.0234 \cdot |\mathcal{C}_{\text{prod}}|$ s. Further, the experiments were set up in the same way as for the centralised algorithm. We did tests for various network sizes (up to 1,225 nodes); see Figure 4.11 for the resulting run times. The run time is mostly due to the processing time at the nodes; the communication overhead was negligible. For comparison, this figure also shows the timing results for the centralised algorithm. Note that the results for the centralised algorithm in Figure 4.11 are for execution on a relatively fast laptop, while the distributed algorithm was



(a) TT task (post-optimisation max: 6,505) (b) SM task (post-optimisation max: 4,073)

Figure 4.9: Comparison of the average memory usage before (dashed lines) and after (solid lines) the optimisations described in Section 4.2. Measured is the average number of values that need to be stored over all steps of the cluster algorithm. The average memory usage of the optimised algorithm is nearly independent of the network size. The results are for networks with 27 configurations per node.

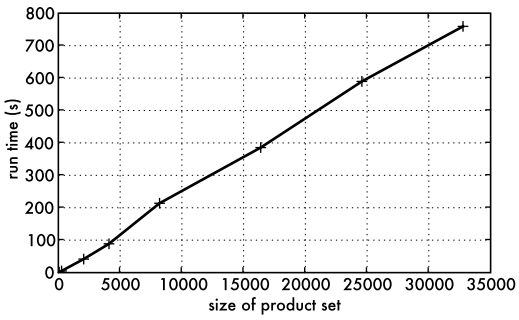


Figure 4.10: Profiling results for a TelosB sensor node. One algorithm step was implemented in TinyOS. Tests were done for input configuration sets of various sizes. The graph shows the run times for various sizes of the product set. We observe a nearly linear relation with slope 0.0234 s per configuration in the product set. We use this information in the simulations.

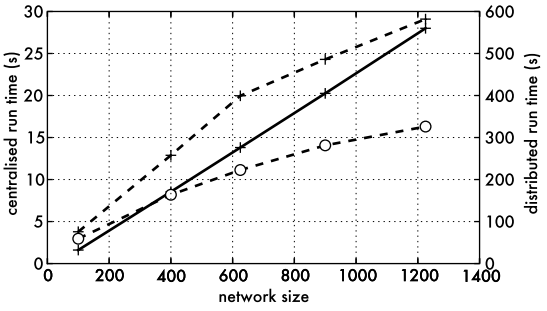


Figure 4.11: Run time of QoS optimisation: centralised (solid lines), distributed (dashed lines) and distributed reduced with a threshold of 750 (dashed lines, ○-markers). Note that the scales for centralised and distributed are different.

Table 4.5: Pareto-set Reduction (900-node networks)

Max. $\lvert\mathcal{C}_{\text{prod}}\rvert$	Run time (s)	Quality loss	Reduction (%)
∞	485.97	0	0
4,000	392.41	0.004	0.1
2,000	355.24	0.011	0.2
1,000	309.20	0.032	1.3
750	281.35	0.065	2.4
500	219.77	0.149	4.7
250	125.05	0.241	10.0

based on TelosB sensor nodes. It turns out that it takes about 8 minutes and 6 seconds on average to do QoS analysis for a 900-node TelosB network with the distributed method, compared to 20 s for the centralised implementation on a powerful laptop. Given the very limited processing power of TelosB nodes, we believe this is reasonable, especially when more powerful nodes are available. Furthermore, the configuration time can be further decreased quite a bit, while giving up a little quality, by the reduction method introduced in Section 4.4, as shown in the next sub-section. And finally, the scalability benefit of parallelisation is visible in this graph: the trend is clearly sub-linear (refer to Equation 4.6).

4.6.5 Complexity Control

Table 4.5 shows the results of experiments with reduction of Pareto sets, for the distributed QoS-optimisation algorithm running on TelosB sensor nodes. The set-up is the same as before. Varied is the maximum allowed size of the product set. When the limit is exceeded, the configuration sets are reduced by removing configurations as explained in Section 4.4. Reduction is done by the heuristic approach (Algorithm 4.7). The first column in Table 4.5 indicates the maximum allowed size of the product set. The next columns show the resulting QoS-optimisation run time, the quality loss L (as in Definition 2.3), and the fraction of nodes that experienced reduction.

We see in the table that the gain in run time is indeed significant, and improves consistently when the threshold on the product-set size is reduced, up to about four times faster for 900-node networks with a threshold of 250. At the same time, the quality loss increases. Since the gain in run time is large, Pareto-set reduction with the heuristic method appears to be useful if we could tolerate some quality loss. Suppose we tolerate a loss of about 6.5%, we would use a threshold of 750 and arrive at an average run time of 281.35 s, an improvement of about 42%. In this case, about 2.4% of the nodes have applied reduction. Figure 4.11 contains run-time results for this threshold, for various network sizes.

Table 4.6: Experimental results for multiple tasks sharing a single configuration space.

SM (%)	TT (%)	SM+TT (%)	Run time (s)	Pareto points [1]	Pareto points [2]
20	20	60	17	16	8
33	33	33	19	18	8
40	40	20	20	19	9
45	45	10	21	19	9
50	50	0	37	19	11
0	90	10	15	15	8
0	0	100	13	13	8
100	0	0	11	9	7
0	100	0	24	13	9

[1] *number of Pareto points for the whole network*
[2] *average number of Pareto points over all clusters*

4.6.6 Multiple Tasks

Finally, we tested Algorithm 4.4 for different combinations of the three node types in one network, sharing a single configuration space (see Section 4.5). The set-up was the same as above (centralised execution, same 900-node networks and 27 configurations). The results are summarised in Table 4.6. Note that the run-times vary quite a bit: in some cases the run times are shorter than in the homogeneous case, sometimes longer. The differences arise from a varying number of Pareto points, as seen in the last two columns of Table 4.6. These numbers are quite unpredictable, since they depend on many factors. The number of Pareto points may change a lot even when small changes to model constants are made. Most importantly, also in these multi-task networks, the run-times do not explode, but remain very short given the complexity of the problem.

4.7 Summary

The algorithms introduced in this chapter belong to the QoS-optimisation phase of the configuration process (phase 3), which is executed after the network has been initialised and the routing tree has been constructed. In this phase, which is the heart of the configuration method, the Pareto-optimal WSN configurations are determined, using the foundation laid in Chapter 3.

Since the configuration space grows exponentially with the size of the network (the number of nodes), trying all possible combinations of parameter values (parameter vectors) is not scalable. The QoS optimiser introduced in this chapter takes advantage of the hierarchical cluster structure of our WSN models. The optimiser starts at the lowest level, in which each cluster contains just a single node, and determines the Pareto points of these clusters. Subsequently, it incrementally

forms larger clusters, while in each step non-optimal configurations are removed. If the number of Pareto-optimal configurations that remains after each step is relatively small, this algorithm is very efficient. The experimental evaluation shows that this is indeed the case for the two WSN tasks introduced earlier. Finding the 9 Pareto points of a 900-node network in the example set-up with 27^{900} potential solutions took a mere 20.2 seconds on a laptop. Moreover, the experiments show a linear relationship between the network size and the algorithm's run time.

We specify conditions for which the algorithm correctly finds all Pareto points of a WSN task, and show that our target-tracking and spatial-mapping tasks comply with these requirements. The order in which clusters are combined appears to be of paramount importance to the correctness. In each step, a leaf cluster must be formed, which means that for each node in the new cluster, also all of its descendants must be in the cluster. Hence, a correct cluster strategy starts at the leaf nodes and progresses towards the root.

We further present a number of techniques to implement the algorithm in a time- and memory-efficient manner. The complexity of a cluster step is reduced, firstly by interleaving several Pareto-algebra operations such that dominated configurations can be removed immediately when found, and secondly by the indexing of used configurations from the inner clusters instead of working with their full parameter vectors. Besides that, quantisation makes use of the fact that the metrics computed by the mapping functions have limited accuracy by ignoring insignificant differences between metric values.

Owing to its incremental leaf-to-root nature, and the above implementation optimisations, it is straightforward to distribute the cluster algorithm. The algorithm is able to execute even on a network of very basic sensor nodes, and finish the 900-node test in about eight minutes.

If the algorithm is still not fast enough, and one is willing to sacrifice some of the task's quality, Pareto-set reduction can be used to set the quality/cost meta trade-off to any desired point. Especially when running the algorithm on resource-constrained sensor networks, restricting the maximum size of the Pareto sets is useful, if not necessary. Simulations show that the run time of algorithm on sensor nodes can be reduced by more than three minutes, if 6.5% of the quality is forfeited.

Finally, we give hints on how to apply this method to QoS optimisation for multiple tasks that simultaneously run on a single (heterogeneous) WSN, and which are the difficulties that need to be overcome. This is an interesting direction for future work.

The main difference with our approach and randomised multi-objective optimisers such as

genetic algorithms, is that we guarantee to find *all* Pareto points for a given model. An experiment also shows the huge speed difference our algorithm has over a genetic algorithm, owing to the smart way of searching through the solution space.

Chapter 5

Routing-Tree Construction

The WSN configuration process introduced in this thesis focuses on networks that employ a routing tree for communication between sensors and the sink. The construction of the routing tree has been factored out of the QoS optimisation phase into a separate phase, such that cluster-based QoS analysis can be performed, which is efficient and scalable. This chapter covers routing-tree construction, phase 2 of the six configuration phases defined in Section 3.4, while Chapter 4 discussed all other phases of the static configuration problem (1, 3, 4, and 5; an integrated experimental evaluation follows in this chapter), and Chapter 6 introduces techniques to tackle run-time dynamism.

The routing tree has an enormous impact on not only the quality metrics of typical sensor-network tasks, but also on the complexity of the QoS optimiser. Important properties of a routing tree are the average path length and the maximum node degree. Ideally, both the average path length and the maximum node degree would be as low as possible. In Section 5.1, we discuss the relevance of these properties. Sections 5.2 and 5.3 introduce centralised algorithms to construct routing trees having various trade-offs between path length and node degree. Section 5.4 gives similar, but distributed algorithms that run directly on the WSN and go hand-in-hand with the distributed QoS optimiser of Section 4.3. Section 5.5 provides an experimental evaluation of the tree algorithms, as well as an overview of all static configuration phases as covered till this point.

5.1 Approach

For a given network, many different (rooted) spanning trees can usually be constructed. For every spanning tree, the attainable quality-metric values (and thus the set of Pareto-optimal

configurations) could be different, and also the configuration time varies. As concluded in Section 4.1.5, it is beneficial for the run time of the QoS-analysis algorithm to have a routing tree in which the node degrees (number of child nodes) are as low as possible. However, minimisation of node degrees when finding a spanning tree of a graph generally conflicts with the desire to have short paths, which gives rise to another trade-off to be taken into account. There are quality metrics in the example models of Section 3.2, such as *detection speed* (the inverse of the maximum event-to-sink delay), that are generally better when paths in the tree are shorter. On the other hand, more-hop paths can also have a better end-to-end reliability, if the per-hop distances (in meters) are smaller. Further, reducing the node degree has a positive effect of load balancing, as the traffic is more evenly distributed among the nodes. This improves the network lifetime, which is defined as the minimum lifetime over all nodes. The trade-off to be made is now not only between quality metrics, but also at a higher level between quality metrics and configuration time: a meta trade-off between the objectives specified in Section 3.3.

Intuitively, a routing tree that is Pareto optimal in the sense of the above-mentioned trade-off between quality and configuration time, is a spanning tree with the property that there is no other spanning tree that has an equal or better configuration time, and an equal or better resulting quality (in terms of Definition 2.4) at the same time. We therefore compare the configuration time and the sets of Pareto points belonging to different spanning trees, and select the best trees.

Because an exhaustive exploration of all spanning trees is infeasible, we consider only spanning trees that have good trade-offs between node degree and path length. We hereby assume that the quality of the task improves if the average path length is reduced while the maximum degree remains constant, and vice versa. We start with shortest-path spanning trees (SPSTs), as often done in the literature on sensor networks, and continue with trees that have lower node degrees, but also longer paths. We further show how to make our tree-construction algorithms distributed.

As said in Section 3.1, we allow all types of node deployments (grid, random) as long as it is possible to form a fully-connected network. However, in dense networks more different spanning trees are possible, which provides more freedom for the algorithms, and hence better results. We do assume that all nodes have similar communication capabilities and that all links are symmetric. In the algorithms in this chapter, we consider a link to be present between two nodes, if they are able to communicate with each other using a medium transmission-power level. This leaves enough freedom for the QoS optimiser to adjust the power levels to either save energy or improve the link's reliability, while the network is dense enough for the tree algorithms to be not too

restricted.

5.2 Low-Degree Shortest-Path Spanning Trees

There are usually multiple SPSTs possible in a network with a given root. Let $\delta(i)$ denote the degree of node i. The first goal as follows.

Definition 5.1 (Minimum-Degree Shortest-Path Spanning-Tree Problem). Given a graph $G = (V, E)$ and root node r, create a shortest-path spanning tree with root r that minimises $\max_{i \in V} \delta(i)$.

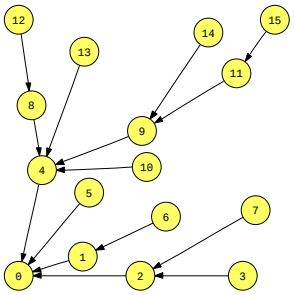
A related and well-known problem in graph theory is that of constructing a minimum-degree spanning tree (MDST). A MDST for a graph is a spanning tree with the smallest maximum node degree, without considering path length or any other costs, and does not have a designated root node.

Definition 5.2 (Minimum-Degree Spanning-Tree Problem). Given a graph $G = (V, E)$, create a spanning tree that minimises $\max_{i \in V} \delta(i)$.

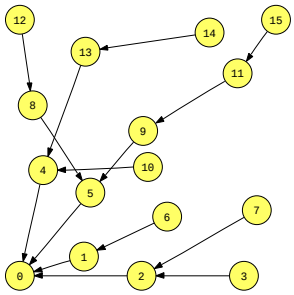
This problem is known to be NP-hard. Fürer and Raghavachari [21] provide an algorithm that yields an approximation of the MDST for undirected graphs. This algorithm is not directly usable for our problem, because of our shortest path requirement and the fact that the algorithm does not take a fixed root node into account. Krishnan and Raghavachari [33] give a similar algorithm for directed graphs with a specified root node, called DMDST. This algorithm still does not optimise for path length, but it can be adapted to serve our purpose.

DMDST starts by constructing an arbitrary spanning tree. Then, the algorithm finds the set S of nodes with the highest node degrees and tries to lower these one by one. If at least one of the nodes in S could be improved, the process starts again; otherwise the algorithm terminates. As the algorithm always improves a node in each step, or halts, it is guaranteed that the algorithm terminates. Improving a node i means finding different routes to the root for one or more of i's children, such that i's degree becomes lower, while other node degrees do not become larger than i's new degree. This effectively balances the node degrees in the tree.

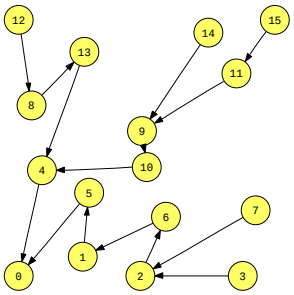
Our adapted algorithm is given as Algorithm 5.1. The function `ContructTree` is called with a directed graph $G = (V, E)$ with V a set of n nodes (equal to $\mathcal{N}$ in our case) and E the set of m



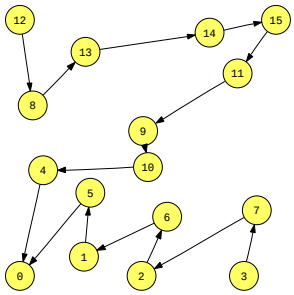
(a) non-optimised:
$\delta_{\max} = 4$, $\delta_{\text{sd}} = 1.3$, $h_{\max} = 4$
$D = 0$, $t = 0.56$ s (centralised), $t = 715$ s (distributed)



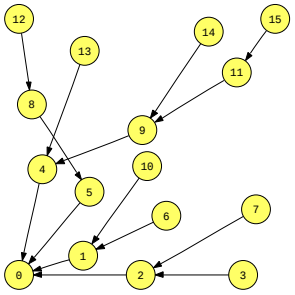
(b) optimised SPST (centralised):
$\delta_{\max} = 4$, $\delta_{\text{sd}} = 1.1$, $h_{\max} = 4$
$D = -0.001$, $t = 0.28$ s



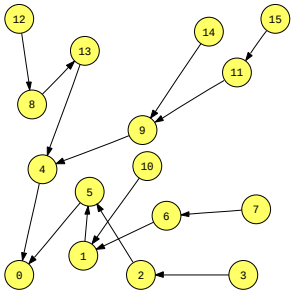
(c) optimised with $\Delta = 2$ (centr.):
$\delta_{\max} = 2$, $\delta_{\text{sd}} = 0.7$, $h_{\max} = 5$
$D = 0.002$, $t = 0.05$ s



(d) fully optimised (centralised):
$\delta_{\max} = 2$, $\delta_{\text{sd}} = 0.4$, $h_{\max} = 9$
$D = 0.012$, $t = 0.03$ s



(e) optimised SPST (distributed):
$\delta_{\max} = 4$, $\delta_{\text{sd}} = 1.1$, $h_{\max} = 4$
$D = -0.001$, $t = 430$ s



(f) fully optimised (distributed):
$\delta_{\max} = 2$, $\delta_{\text{sd}} = 0.9$, $h_{\max} = 4$
$D = 0.001$, $t = 233$ s

Figure 5.1: Examples of trees generated by the various algorithms ($\delta_{\max}$: maximum node degree, δ_{sd}: standard deviation of the node degree, $h_{\max}$: maximum hop count, D: quality difference with non-optimised case, t: run time of tree construction plus QoS optimisation), see Section 5.5 for the experimental set-up.

Algorithm 5.1: SPST construction with balanced node degrees. The function `ConstructTree` *is called with the graph G and a root node r. The operator* $\arg\min$ *is defined to return a set of minimisers as in* (5.1)

```
1  function ContructTree(G, r):
2      T ← BFS(G, r)
3      repeat:
4          δ_max ← max_{i∈V\{r}} δ(i)
5          S ← {i ∈ V\{r} | δ(i) ≥ δ_max − 1}
6          n ← 0
7          for each i ∈ S:
8              n ← n + Improve(G, T, i)
9          if n = 0:
10             return T
11
12 function Improve(G, T, i):
13     ℓ ← 0
14     for each child j of i:
15         C ← {x ∈ V | x neighbour of j in G, h(x) = h(i)  and δ(x) ≤ δ(i) − 2}
16         if C ≠ ∅:
17             change parent of j in T to one in arg min_{m∈C} δ(m)
18             ℓ ← ℓ + 1
19     return ℓ
```

links (there is a link from node A to node B if B is within the communication range of A), and root node r. In the pseudo code, $\delta(i)$ is the degree of node i, and $h(i)$ is the distance in hops (hop count) of node i to the root. The initial tree is an SPST (constructed by a Breadth-First Search (BFS) [13]), and every improvement step maintains the shortest-path property. This means that if a node j is appointed another parent node, the new parent needs to have the same distance to the root as the old one. This rule also ensures that the transformation does not introduce loops and thus maintains a tree: the new parent can never be a descendant of j, since it is one hop closer to the root than j. Moreover, a node only changes its parent, if the new parent has a degree at least two less than the current parent, such that the degrees of both parents become more balanced. Note that the degree of the root node cannot be reduced in an SPST. We therefore exclude the root in lines 4 and 5 of the algorithm; it is not meaningful in line 5, and leaving it in in line 4 would lead to a potentially earlier termination of the algorithm (and a less balanced tree), as nodes with degrees lower than the root are not optimised. Line 5 selects the nodes with large node degrees ($\delta_{\max}$ and $\delta_{\max} - 1$) for balancing.

The function arg min in line 17 is defined to return a set of minimisers, as follows for a function f over a domain $\mathcal{X}$:

$$\arg\min_{x\in\mathcal{X}} f(x) = \{x \in \mathcal{X} \mid f(x) = \min_{x'\in\mathcal{X}} f(x')\} \tag{5.1}$$

An example is given in Figure 5.1. Compare the graphs in Figures 5.1(a) and 5.1(b), respectively showing the non-optimised and optimised trees with the shortest-path constraint. We see two nodes with degree 4 in the non-optimised tree: nodes 0 and 4. In the optimised tree, two of node 4's children have been moved to node 5 to eliminate the high degree. Furthermore, the degree of node 9 has been lowered from 2 to 1. However, the root's degree could not be reduced due to the shortest path constraint. If we execute the QoS-optimisation algorithm with 27 configurations per node as in Section 4.6, the quality difference D between the resulting Pareto sets of the non-optimised and optimised tree arrives at -0.001 (i.e. -0.1%). Thus, the SPST-optimised tree gives slightly better results. More significant is the gain in run time (tree construction plus QoS optimisation, which goes down from 0.56 s to 0.28 s. Thus, it is clear that degree optimisation is useful.

The complexity of Algorithm 5.1 depends on the number of improvement steps and the complexity of the improvement function. The latter is a loop over all children of a node i, in which each iteration takes constant time. Krishnan and Raghavachari [33] showed by experiment that the number of improvement steps grows approximately linearly with the number of nodes in the network. Thus, if $\delta_{\max}$ is the largest node degree in the initial tree constructed by BFS, the practical time complexity of the degree-improvement is $\mathcal{O}(\delta_{\max} \cdot n)$. Furthermore, the initial BFS runs in $\mathcal{O}(n \cdot m)$.

5.3 Node-Degree and Path-Length Trade-offs

Reducing the node degree even more can only be done by making paths longer. Since there are generally few nodes with a very high node degree, it is expected that not many paths need to be enlarged to attain a significant improvement. We wish to solve the following problem.

Definition 5.3 (Degree-Constrained Shortest-Path Spanning Tree Problem). Given a graph $G =$

Algorithm 5.2: Tree construction with balanced node degrees; no shortest-path constraint. The function `ConstructTree` *is called with the graph G, root node r, and a degree target Δ as its arguments. The operator* $\arg\min$ *is defined to return a set of minimisers as in* (5.1).

```
1  function ContructTree(G, r, Δ):
2      T ← BFS(G, r)
3      repeat:
4          δ_max ← max_{i∈V} δ(i)
5          S ← {i ∈ V | δ(i) ≥ δ_max − 1 and δ(i) > Δ}
6          n ← 0
7          for each i ∈ S:
8              n ← n + Improve(G, T, i, Δ)
9          if n = 0:
10             return T
11
12 function Improve(G, T, i, Δ):
13     ℓ ← 0
14     for each child j of i:
15         C ← {x ∈ V | x neighbour of j in G, x no descendant of j and
               δ(x) ≤ δ(i) − 2}
16         if C ≠ ∅:
17             change parent of j in T to ChooseParent(C, Δ)
18             ℓ ← ℓ + 1
19         break if δ(i) ≤ Δ
20     return ℓ
21
22 function ChooseParent(C, Δ):
23     S ← {i | δ(i) < Δ, i ∈ C}
24     if S = ∅:
25         S ← arg min_{i∈C} δ(j)
26     return arbitrary element of arg min_{i∈S} h(i)
```

(V, E), root node r and degree target Δ, create a spanning tree with root r that minimises

$$\frac{1}{|V|} \sum_{i \in V} h(i), \tag{5.2}$$

subject to

$$\delta(i) \leq \Delta, \quad \text{for all } i \in V. \tag{5.3}$$

Since this MDST problem is NP-hard, also this problem is intractable.

DMDST is an algorithm that optimises for node degree as much as possible, so it almost does what we need. The difference is that we do not need to optimise degrees beyond a given degree target Δ, only path lengths. A simpler trade-off algorithm is obtained by slightly altering Algorithm 5.1, and including the parameter Δ; the adapted algorithm is given as Algorithm 5.2, and explaining in the remainder of this section.

In the trade-off algorithm, the hop-count condition in line 15 of Algorithm 5.1 is removed to enable longer paths as well. Instead, we need another condition in line 15. Since it is now possible that a candidate new parent in C is a descendant of j, changing to such a parent creates a loop. This needs to be verified by following the path from the candidate parent to the root: if j is not on the path, no loop would be formed and the parent can safely be chosen. As also the root's degree can now be lowered, we remove the exclusion of the root from lines 4 and 5. DMDST allows the new path from a node j to the root to initially go through the sub-tree of j. This may lead to smaller node degrees (and longer paths), but involves a BFS for each improvement step, and is therefore more complex than our algorithm.

In line 17 we no longer pick the candidate parent with the lowest degree, since optimising beyond degree Δ is not needed. The function `ChooseParent` is introduced, which selects, from a set C, a parent that has a shortest path to the sink and a degree at most Δ, or otherwise having the lowest degree available.

Finally, to establish control on the trade-off between path length and node degree, a stop condition is built in: only nodes with a degree more than Δ are attempted to be improved. This leads to the extra term $\delta(i) > \Delta$ in line 5 of Algorithm 5.2, in which the sets of candidate nodes for optimisation is formed. Furthermore, the loop in the `Improve` function should be stopped when $\delta(i) \leq \Delta$. Since a solution to the problem of Definition 5.3 may not exist (and if it exists,

we may not find it due to the intractability of the problem), the algorithm may terminate without meeting constraint (5.3).

The impact of these changes can be seen in Figures 5.1(c) and 5.1(d). Compare Figure 5.1(a) of the initial tree with Figure 5.1(c), in which all nodes with a degree more than 2 are optimised ($\Delta = 2$), regardless the path length. We see that now also the root's degree has been reduced, such that the largest node degree is lowered from 4 to 2, but there are still four nodes with the maximum degree of 2. The degree improvement is at the expense of an increase of one hop in the longest path. In the fully degree-optimised tree in Figure 5.1(d), all nodes except the root have degree 0 or 1, but the longest path is now 9 hops long. The quality differences of the Pareto sets resulting from the non-optimised tree, with the optimised tree with $\Delta = 2$ and the fully-optimised tree respectively, amount to 0.002 and 0.012, so now the results are slightly worse. On the other hand, the run time of tree construction plus QoS optimisation is a lot lower: an improvement from 0.56 s to 0.05 s and 0.03 s. Here it seems that especially degree optimisation with $\Delta = 2$ has a very good trade-off between run time and quality.

5.4 Distributed Tree Optimisation

Since the degree-improvement steps in Algorithm 5.1 and Algorithm 5.2 use only information from nodes in the neighbourhood of the node being improved, it is possible to use a similar mechanism in a distributed degree-reduction algorithm. The main difference is the selection of the node to be improved next, which is based on global knowledge in the centralised algorithms. The use of global knowledge is infeasible in a distributed algorithm. Furthermore, we need to take the unreliable nature of wireless communication into account and design a robust algorithm.

A state diagram of the distributed algorithm is shown in Figure 5.2. Together with Figure 4.6, this figure forms the state diagram for phases 1 through 5 of the distributed configuration process. This program runs in each node, except the root, which starts directly in the *parent set* state. A state change may occur upon reception of a message from another node, or when a timer expires. These events may also trigger actions. State changes and associated events and actions are drawn as arrows in the diagram. A node should eventually have a single parent, and a number of children. Each node that is within communication range and is not a parent or child is called a peer.

In the initial *no parent* state, a node does not have a parent, and is therefore not yet admitted to the tree. Flooding from the root node, the distributed equivalent of a breadth-first search, is used

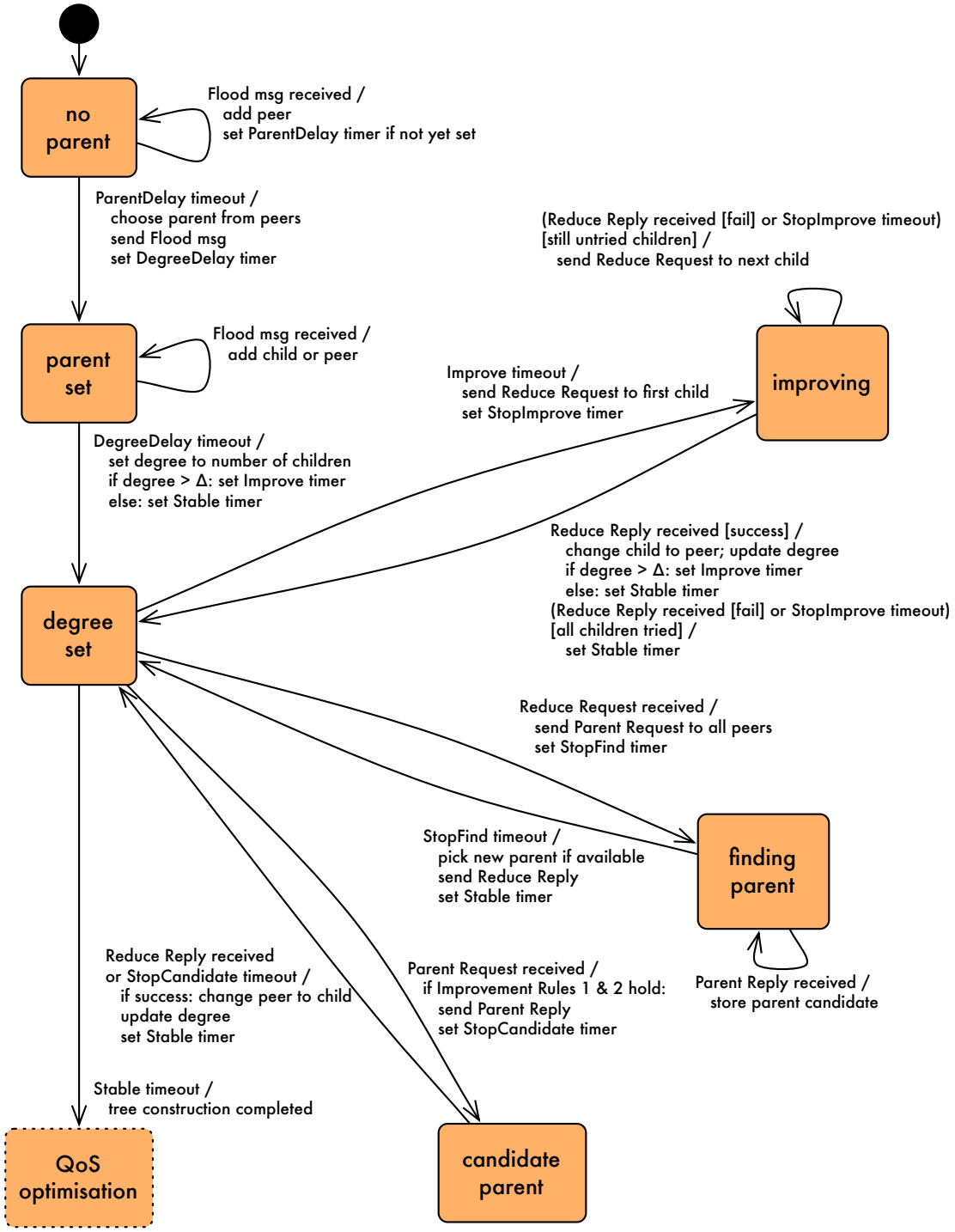


Figure 5.2: Distributed tree construction, state diagram. State transitions are triggered by events and/or conditions as annotated at the arrow before the slash. An event can be due to an incoming message from another node or a timer expiry. Actions at a transition are given after the slash. All events that are not listed at a state are ignored. Timers are local to a state; leaving a state implies resetting a timer.

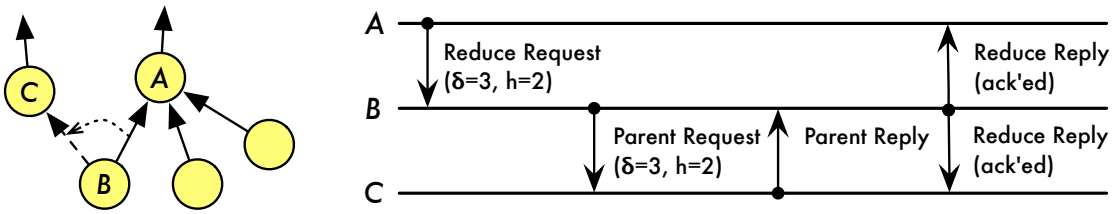


Figure 5.3: A degree-improvement step.

to set up the initial spanning tree. The root node initiates the process by broadcasting a Flood message. A Flood message contains the ID of the sender and its parent, as well as its hop count. Upon receiving the first Flood message from a candidate parent, a node waits for a short while to collect more messages from potential parents. To this end, it sets the ParentDelay timer. This delay is used, because in a practical network, the message that arrives first is not necessarily from the node with the shortest hop count. Following the delay, it chooses a candidate with the smallest hop count to the root as its parent, broadcasts a new Flood message, and enters the *parent set* state. This is an intermediate state intended to detect the node's children – and thus its degree – and other peers by overhearing messages. Imperfect overhearing can be corrected by making a parent acknowledge a new child, and the child retransmitting the message if needed. The *degree set* state is reached after the DegreeDelay timer expires, and the neighbourhood of the node is deemed to be stable. At nodes closer to the sink, this happens earlier than further down the tree, but owing to the locality of the algorithm, this makes no difference. This is the end of the flooding phase, and degree reduction commences.

The node-degree balancing scheme is initiated through a mechanism of timers. If the degree of a node is higher than the degree target Δ, it sets the Improve timer with a duration inversely proportional to the degree, and starts the improvement procedure only after expiration of this timer. This ensures that high-degree nodes are improved first. Time synchronisation is not needed in the network, because the algorithm uses completely local handshakes between three nodes, as explained below. Such handshakes may happen concurrently at any time and place in the network.

Refer to Figure 5.3 for the following explanation. Improvement starts with a Reduce Request message from node A, which wants to reduce its degree after expiry of its Improve timer, to node B, one of its children. This places node A in the *improving* state, and B in the *finding parent* state. B will then attempt to find another parent node. It does so by broadcasting a Parent Request message that its peers would receive. Both messages contain the degree of A and its distance to

the root. A peer C checks whether it is indeed a suitable new parent by comparing its node degree and distance to the root with those of the current parent A (see below for the precise conditions). If it is, C answers with a Parent Reply message to B, and changes to the *candidate parent* state. After a delay during which other candidate parents may reply, B will change its parent to a candidate parent according to the function `ChooseParent` of Algorithm 5.2, C in this example. B confirms the change to both A and C by sending a Reduce Reply message, such that they could update their child and peer lists, and degree. After that, A may schedule another Improve timer if the degree is still higher than the threshold Δ. If B does not manage to change its parent (a timeout occurs), node A repeats the procedure with another child if possible, or gives up. Following this procedure, the three nodes return to the *degree set* state. When entering this state with a sufficiently low degree, the Stable timer is started. The tree-construction algorithm completes at a node when this timer expires without interruption; the timer is reset when leaving the *degree set* state. At this point, the node's parent and children are fixed and QoS optimisation is started (see Chapter 4).

Due to message loss during the flooding phase, it is possible that some nodes are not admitted to the tree. In dense networks – and WSNs are usually dense – the chances of this happening are quite low, because each node has many potential parents, and thus flooding has a lot of inherent redundancy. Additional measures can be taken to ensure that nodes that missed all flooding messages still find a parent node. Handshaking between A, B and C is used to properly update the parent, child and peer data of all nodes in the improvement phase, as visualised in the timing diagram of Figure 5.3 and the state diagram of Figure 5.2. B's confirmation to A and C should be acknowledged, and retransmitted if needed. If A or C still do not receive the confirmation, their estimation of the degree would be incorrect. The consequence of this is that further degree-optimisation may be incorrect, or oscillations occur (nodes switch parents indefinitely). The latter can be avoided by setting a limit on the number of switches a node can make. An incorrect list of children, however, never breaks the tree, which is defined only by the parent variable in each node. As there are typically few nodes with high degrees (the average degree in a tree is a constant), the cost of the algorithm stays limited. Finally, the correct behaviour of the algorithm depends on properly set timer values. See Table 5.1 below in the experimental section, for an overview of the timer values used in our experimental set-up.

As before, node C can decide in two ways whether it is a suitable new parent, depending on whether path length has a higher priority than node degree or not. Either way, C's degree should be at least two less than the degree of A:

Improvement rule 1.

$$\delta(C) \leq \delta(A) - 2 \tag{5.4}$$

If an SPST is required, as before, the following additional rule applies, which ensures that path lengths remain the same and no loops are introduced:

Improvement rule 2(a) (Shortest-path constrained).

$$h(C) = h(A) \tag{5.5}$$

In the case without shortest-path constraints, the centralised Algorithm 5.2 ensures that C is not a descendant of B by following the path from C upwards, in order to prevent loops from being formed. This is a relatively expensive operation in the distributed case, as a lot of messages may be needed, and not entirely trustworthy because of the unreliable wireless communication. We suggest a compromise that poses extra requirements on candidate parents based on the proposition below. Further, when the hop count of a node changes, also the hop count of all its descendants in the tree changes. Updating the distance state in each descendant would require a message to be propagated all the way down to the leaves. For robustness and energy considerations, however, we choose to keep the algorithm localised, and therefore do not update the hop-count variables. Only in the loading phase, the hop counts are updated again for later use. The consequence is that not all potential for degree reduction is used. Experiments in Section 5.5 show that our approach still leads to a large improvement in node degree.

Proposition 5.1 (Loop-freeness). *Let $h(i)$ be the hop count of node i to the root in the* initial *tree; $h(i)$ is not updated when the tree is changed. We impose the following requirements on a candidate parent C of a node B with current parent A:*

1. $h(C) \leq h(B)$
2. $h(C\text{'s parent}) < h(C)$

As a result, loop-freeness is guaranteed.

Proof. Requirement 1 ensures that the nodes on any path in the tree starting from the root are ordered by ascending h. For example:

```
0  <- 1  <- 2  <- 3  <- 4  <- 5  <- 6
h=0   h=1   h=1   h=2   h=2   h=2   h=3
```

A loop can only be formed if a node connects to a node further down on such path (a descendant in the tree). Because of 1, a node cannot connect to a node with a higher h. Hence, loops can only be formed when connecting to a node with the *same* h (node 3 connects to 5 in the example). Requirement 2 allows a node to only connect to the first node with a specific value of h on a path, e.g. 0, 1, 3, or 6. This ensures that a node B cannot connect to a node C with the same h if C is a descendant of B, and thus eliminates the possibility of loops. □

The requirements in Proposition 5.1 can be made a little looser, without changing the reasoning in the proof, which leads to the following rule:

Improvement rule 2(b) (Unconstrained).

$$h(C) < h(B) \text{ or } (h(C) = h(B) \text{ and } h(C\text{'s parent}) < h(C)) \tag{5.6}$$

This allows node 4 to connect to node 2 in the example of the proposition's proof.

Proposition 5.2 (Tree property). *An improvement step that follows rule 1 plus rule 2(a) (for an SPST) or 2(b) (no path-length constraints) does not break an existing tree.*

Proof. A graph is a tree if every node (except the root) has exactly one parent node, and the graph contains no loops. An improvement step may update the parent of a node B from node A to C, if C allows this, ensuring that B again has a valid parent. Improvement rules 2(a) (trivial) and 2(b) (similar reasoning as in the proof of Proposition 5.1) ensure loop-freeness in the SPST and non-SPST cases. □

In the example of Figure 5.1, the distributed algorithm with shortest path restriction (Figure 5.1(e)) leads to a tree with maximum node degree $\delta_{\max} = 4$, standard deviation of the node degree $\delta_{\text{sd}} = 1.1$, and maximum hop count $h_{\max} = 4$. The unrestricted version (Figure 5.1(f)) arrives at $\delta_{\max} = 2$, $\delta_{\text{sd}} = 0.9$, and $h_{\max} = 4$, which is similar to the results of the centralised algorithm with $\Delta = 2$. The quality differences between the resulting Pareto sets from the non-optimised tree, and both optimised trees are -0.001 and 0.001 respectively. The run times of

Table 5.1: Timer values for distributed tree optimisation

Timer name	Value (s)
ParentDelay	0.01
DegreeDelay	0.10
Improve	$\frac{1}{4\delta}$
StopImprove	0.12
StopFind	0.05
StopCandidate	0.12
Stable	0.50

distributed tree construction plus distributed QoS optimisation (see Section 4.3) improve from 715 s to 430 s and 233 s in the respective cases.

5.5 Experiments

In this section we present experimental results on the various aspects of centralised and distributed routing-tree creation. These experiments cover tree creation with and without shortest-path constraints, and various values of Δ for the latter. The experiments were set-up in the same way as in Section 4.6. The effect of reducing node degrees in a shortest-path spanning tree was tested on 100 networks of 900 nodes each, randomly deployed in an area of 300 $\times$ 300 m, and communication ranges of 20 m. This time, however, no restrictions on the node degree were applied; the previous experiments only used networks with node degrees of at most 3, which were actually generated by Algorithm 5.2 with $\Delta = 3$. The simulations to test the distributed algorithm take packet loss due to random bit errors and collisions into account. The timer values that were used in the distributed algorithm are listed in Table 5.1. The results are presented in Section 5.5.1.

Since at this point we have covered all static configuration phases (phase 1 to 5), which lead to a properly configured WSN, we present and overview of results of these phases. It is particularly interesting to observe the differences between the centralised and distributed approaches. See Section 5.5.2 for these results.

5.5.1 Tree Optimisation

For all 100 networks, routing trees were constructed by both a simple breadth-first search or flooding, to serve as reference algorithms that construct SPSTs without degree reduction, and the centralised and distributed degree-reduction algorithms introduced in this chapter. The run times

Table 5.2: Results on tree construction: node degree and hop count (averages over 100 900-node networks)

Centralised

Node-degree optimisation	Node degree max	Node degree st.dev.	Hop count max	Hop count mean (st.dev.)
Non-optimised	6.06	1.14	26.43	14.65 (0.25)
Optimised SPST	5.14	0.88	26.43	14.65 (0.25)
Fully optimised	2.00	0.33	52.25	26.01 (2.51)
$\Delta = 2$	2.00	0.75	34.95	18.75 (1.12)
$\Delta = 3$	3.00	0.99	29.25	16.22 (0.72)
$\Delta = 4$	4.00	1.07	27.04	15.12 (0.55)
$\Delta = 5$	5.00	1.09	26.62	14.80 (0.38)

Distributed

Node-degree optimisation	Node degree max	Node degree st.dev.	Hop count max	Hop count mean (st.dev.)
Non-optimised	7.10	1.33	26.71	14.76 (0.25)
Optimised SPST	5.21	0.92	26.75	14.76 (0.25)
Fully optimised	3.08	0.80	32.19	17.42 (0.79)

were recorded, as well as the node degrees and path lengths of the trees. Subsequently, the cluster algorithm for QoS optimisation was executed for all trees (target-tracking task, 27 configurations per node), and the run times were recorded. Also determined was the quality difference of the Pareto set for the degree-optimised tree compared to the results from the reference trees, based on Definition 2.4. The results are given in Tables 5.2 and 5.3. In each table, the first row is for SPSTs without degree optimisation, the second is for degree-optimised SPSTs, and the remaining rows are for further degree optimisation given a degree target Δ, while allowing paths to grow longer. Here, the "fully optimised" algorithm balances node degrees as much as it can. The distributed algorithm functioned properly in all cases, even in the presence of packet loss: all nodes were correctly included in the tree.

First of all, Table 5.2 shows that the node-degree optimisation algorithms, both centralised and distributed, really do what they are supposed to do: reducing high node degrees. It is evident from the reduction in the standard deviation of the node degree that the degrees are more balanced after optimisation. The average node degrees are not listed in the table, as for any tree of n nodes, the mean node degree is a constant equal to $\frac{n-1}{n}$. In the SPST case, the maximum node degree is not lowered much by optimisation. The maximum degree in the SPSTs was often at the root node, which cannot be reduced, since all one-hop neighbours of the root have to be its children in an SPST. However, the standard deviation did become lower, which already has an

Table 5.3: Results on tree construction: run times and quality (averages over 100 900-node networks)

Centralised

Node-degree optimisation	Tree-construction run time[b] (s)	QoS-optimiser run time[b] (s)	Slower[a] (%)	Quality diff[b,c]
Non-optimised	0.013 (0.000)	666.41 (830.79)	n/a	n/a
Optimised SPST	0.015 (0.001)	303.16 (629.93)	11	-0.01 (0.02)
Fully optimised	0.717 (0.046)	4.11 (0.13)	0	0.08 (0.03)
$\Delta = 2$	0.316 (0.029)	5.69 (0.15)	0	0.04 (0.03)
$\Delta = 3$	0.097 (0.017)	20.25 (1.75)	0	0.01 (0.03)
$\Delta = 4$	0.034 (0.009)	62.78 (14.02)	0	0.01 (0.03)
$\Delta = 5$	0.019 (0.004)	158.62 (72.66)	10	0.00 (0.02)

Distributed

Node-degree optimisation	Tree-construction run time[b] (s)	QoS-optimiser run time[b] (s)	Slower[c] (%)	Quality diff[a,b]
Non-optimised	0.386 (0.011)	$5.66{\cdot}10^4$ ($2.20{\cdot}10^5$)	n/a	n/a
Optimised SPST	1.302 (0.199)	$1.33{\cdot}10^4$ ($3.81{\cdot}10^4$)	18	0.00 (0.01)
Fully optimised	1.001 (0.082)	485.97 (366.08)	1	0.01 (0.01)

[a] Relative number of times the optimiser run time for the optimised tree is *worse* than without degree optimisation.
[b] Values are averages over all networks, with the standard deviation given in brackets.
[c] $D(\mathcal{C}_\mathrm{R}, \mathcal{C}_\mathrm{S})$ as in Definition 2.4, with $\mathcal{C}_\mathrm{R}$ the Pareto set without degree optimisation, and $\mathcal{C}_\mathrm{S}$ the Pareto set with degree optimisation.

effect on the following configuration phase, QoS optimisation. If paths may be made longer, also the maximum node degree can be reduced significantly: from about six to two (centralised), or seven to three (distributed). The increase of the longest path, however, is significant.

Note that the distributed algorithms perform a little worse compared to the centralised algorithms. This is explained by the presence of packet loss and the variable delay of packets travelling in a wireless network. Also, we willingly gave up some potential for degree reduction in the distributed algorithm in order to keep it localised (see Section 5.4). The hop counts of "SPSTs" created by the distributed algorithm, however, are only a bit longer than the real SPSTs constructed under ideal circumstances. Regarding the node degree, the performance of fully-optimised distributed tree construction is comparable to optimisation with $\Delta = 3$ in the centralised case.

From Table 5.3, it is clear that gain in speed of the QoS optimiser outweighs the extra run time needed for node-degree optimisation. The QoS optimisation time can be reduced by about 162 times to just over four seconds in the centralised case, and about 116 times for the distributed algorithms. The smallest reduction can be seen when forcing the tree to be an SPST. However, even though the SPST has better-balanced node degrees, the standard deviation of the run times

is very large, implying that the times vary widely. More importantly, we also see in Table 5.3 that for 11% of the networks (18% for distributed), node degree optimisation with an SPST restriction actually makes the QoS optimiser run longer than before. This is attributed to the unpredictable nature of the number of Pareto points of a cluster. Changing the tree may increase the number of Pareto points of a cluster so much that, even if node degrees are lower, QoS optimisation overall takes more time. This becomes even worse due to the fact that, in an SPST, the degree of the root node can never be reduced. Without path-length constraints, the node degrees can be reduced much more, and these effects diminish. In the fully-optimised case, the QoS optimiser's run time has a much smaller standard deviation. It is interesting to note that full degree-optimisation in the distributed case takes less time than SPST-constrained tree optimisation. This is probably owing to the additional freedom the unconstrained algorithm has over the SPST-algorithm, which leads to quicker results, even though more (concurrent) work is done.

Next, consider the last column of Table 5.3, which lists the quality differences between the Pareto points for the non-degree-optimised trees and the optimised versions. It appears that the degree-optimised SPST is sometimes better than the non-optimised tree and sometimes worse, but on average they lead to Pareto sets of similar quality. Further optimised trees tend to yield Pareto sets of slightly lower quality, but the differences are quite small. Coupled with the enormous gain in QoS-optimisation run time, it is clear that – at least for the target tracking WSN model used here – node degree reduction is very useful, even if the paths from sensors to the root are increased.

For the remainder, we use centralised tree optimisation with $\Delta = 3$ and fully-optimised distributed tree optimisation, since we consider these to have good quality/configuration time trade-offs. Moreover, these instances of the centralised and distributed tree algorithms show similar performance.

Figure 5.4 shows what happens to the run times of the centralised and distributed tree algorithms for different network sizes. It is clear that the distributed algorithm is much better scalable. The communication overhead causes it to be a little slower than the centralised execution, but the run time does not increase a lot when the network size grows, while the run time of the centralised algorithm increases more than linearly with the network size.

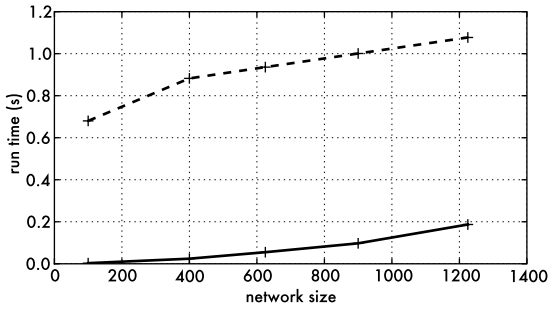


Figure 5.4: Run time of tree-construction: centralised (solid lines) and distributed (dashed lines).

5.5.2 The Complete Configuration Process

An overview of the costs of phases 1, 2, 3 and 5 of the configuration process for 900-node networks, for the centralised and distributed approaches, as well as for distributed with reduction (product-set threshold of 750), is given in Table 5.4. This table includes for every phase the total time it takes to complete, the average processing time per node, and the average number of bytes transmitted per node. Furthermore, Figure 5.5 illustrates the total run time for varying network sizes. This graph includes the results for QoS optimisation from Figure 4.11, which dominate the total run time. We do not consider the details of selecting a specific configuration from the Pareto-optimal set (configuration phase 4), since this selection is application specific. Since the number of Pareto-optimal configurations is typically very small, the selection phase will have little or no impact on the costs of the configuration process.

Looking at only the initialisation, tree construction, and loading phases together, we see that the distributed implementation is much faster than the centralised implementation, and that it has in total a significantly smaller communication overhead as well. The QoS-optimisation phase, however, takes much longer when executed on the sensor nodes directly, as we used nodes with very limited processing capabilities (TelosB, see Section 4.6). Therefore, the centralised approach has the best overall run time, while it still has a larger communication overhead for the nodes. The run time of the distributed QoS optimisation can be improved by the Pareto-set reduction techniques of Section 4.4. Furthermore, from Figure 5.5 it is clear that the distributed approach is better scalable to large networks.

Table 5.4: Configuration overview (900-node networks)

Phase	Costs[1]	Centralised	Distributed	Distr. reduced
Initialisation	Total time (s)	3.56	0	0
	Processing (s)	0	0	0
	Comm. (bytes)	196.9	0	0
Tree construction	Total time (s)	0.10	1.00	1.00
	Processing (s)	0	0	0
	Comm. (bytes)	0	58.9	58.9
QoS optimisation	Total time (s)	20.25	485.97	281.35
	Processing (s)	0	2.34	1.97
	Comm. (bytes)	0	69.6	70.8
Loading	Total time (s)	3.80	0.45	0.44
	Processing (s)	0	0	0
	Comm. (bytes)	135.0	21.4	21.3
Total	Total time (s)	27.71	487.42	282.80
	Processing (s)	0	2.34	1.97
	Comm. (bytes)	331.9	149.9	151.0

[1] Processing and communication costs are averages per node.

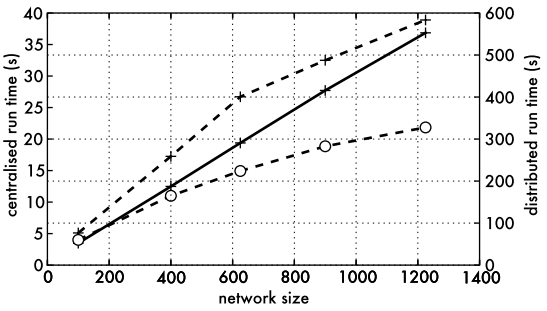


Figure 5.5: Total configuration run time: centralised (solid lines), distributed (dashed lines) and distributed reduced (dashed lines, ○-markers). Note that the scales for centralised and distributed are different.

5.6 Summary

Before the QoS-optimisation phase that is detailed in the previous chapter can be executed, a routing tree needs to be formed. This tree has two properties that impact not only the task quality, but also the complexity of the QoS optimiser. First of all, the average path length of the tree, measured over the paths from each node to the sink, should be as low as possible, as quality metrics such as delay and reliability typically benefit from this.

Secondly, the maximum node degree in the tree should be as low as possible. Ideally, each node has the same, low degree, such that the workload is evenly distributed over the network. This avoids bottlenecks and has a positive impact on the network's lifetime. Moreover, the complexity of the QoS optimiser rapidly increases with the maximum node degree. Balancing degrees across the network further takes the most out of the available parallelism when the optimiser is run on the WSN itself in a distributed fashion.

This chapter introduces centralised as well as distributed algorithms to construct a routing tree with a good trade-off between path length and node degree, which are conflicting objectives. We provide two different optimisation strategies: one in which the maximum node degree is minimised while forcing the tree to have the shortest paths possible, and another in which the average path length is minimised within a maximum-degree constraint.

Experiments show that the algorithms indeed establish a range of degree/path-length trade-offs. More importantly, this also leads to the expected trade-off between task quality and configuration cost, which is tunable by setting the degree target. Hence, together with the complexity control techniques of Chapter 4, routing-tree construction provides powerful means to choose a suitable point on the quality/cost trade-off curve.

The chapter finally provides a roundup on the full configuration process from initialisation to loading. The pros and cons of the centralised versus the distributed approaches are highlighted.

Chapter 6

Run-Time Adaptation

Chapters 4 and 5 completely describe the configuration process for a static network. Wireless Sensor Networks, however, are often dynamic. For example, nodes may run out of battery or move, or the environment changes, such that the wireless connections behave differently. If the network changes, the previously computed configuration is not likely to be optimal anymore. Or worse, the network is broken and some nodes are disconnected from the sink, or quality constraints are violated. It is therefore necessary to be able to adapt to such dynamism at run time.

The most straightforward way to update the configuration is to run the whole configuration process again. This chapter provides methods to more efficiently compute and load a new configuration. There is an important trade-off between the quality achieved by the running task and the cost of configuring the network (see Section 3.3). This trade-off becomes even more critical when regularly adapting to changes in the network.

It is important to consider the granularity and frequency of reconfiguration. Adapting the configuration to all high-frequency fluctuations of the system would not be feasible. Because of scalability issues, it is best to react to such dynamism on a small, local scale using appropriate techniques depending on the application. One could for example temporarily increase the sample rate or duty cycle based on the predicted trajectory of a target [48], or adjust the transmission power in response to fluctuations of the link quality or workload [12].

Such small-scale techniques are orthogonal to our method; the configuration that we establish should be seen as a relatively long-term set point of the node parameters. The computed local metrics should be maintained on average in order to achieve a globally good performance, while short fluctuations are allowed. If more structural changes happen, such as nodes that move, enter or leave the network, a reconfiguration according to our framework should be done to update

the parameters of a possibly large group of nodes, or even the whole network, depending on the desired quality/cost trade-off. Extending the static configuration method of the previous chapters with a variety of reconfiguration techniques to handle coarse-grained dynamism is the topic of this chapter.

Section 6.1 does the ground work by defining the types of events that may occur at run time, and it explores what is needed to adapt to them. Section 6.2 then introduces ways to adapt the routing tree in case the event calls for it. In Section 6.3, we elaborate on tree reconstruction for a specific type of dynamism: the mobile sink. Subsequently, Section 6.4 explains how to optimise the task quality in the new situation, especially in a localised way. Section 6.5 provides an experimental evaluation of the proposed reconfiguration schemes. Finally, Section 6.6 presents another example of how the configuration method can be used.

6.1 Preliminaries

The objectives in Section 3.3 state that the WSN should always be in a configuration $(\bar{p}, \bar{u}, F_{\mathrm{q}}(\bar{p} \cdot \bar{u}), F_{\mathrm{r}}(\bar{p} \cdot \bar{u}))$ that satisfies the constraints and optimises the value function, as specified by the following equations (repeated for convenience):

$$F_{\mathrm{q}}(\bar{p} \cdot \bar{u}) = \min(val(\{F_{\mathrm{q}}(\bar{p}' \cdot \bar{u}) \mid \bar{p}' \in \mathcal{S}_{\mathrm{Pc}}\} \cap \mathcal{D}_{\mathrm{q}})), \tag{3.5}$$

$$F_{\mathrm{r}}(\bar{p} \cdot \bar{u}) \in \mathcal{D}_{\mathrm{r}}. \tag{3.6}$$

The approach that we take in this thesis is to first create a routing tree (determine the parent parameters), and subsequently optimise the remaining parameters. The optimisation problem is specified given a vector of uncontrollable parameters $\bar{u}$. One uncontrollable parameter is the location parameter, which is always present in each node, and this parameter is most important for the tree construction process.

The configuration process described in Chapters 4 and 5 assumes a static situation: $\mathcal{S}_{\mathrm{Pc}}$ and $\bar{u}$, as well as the optimisation criteria (value function and constraints) are given and fixed. It furthermore starts from an unconfigured network. During the lifetime of the task, however, events may occur that cause a change in the situation, and would necessitate a re-evaluation of the above equations, and possibly an alteration of the current configuration.

Definition 6.1 (Events). We distinguish the following types of events:

1. A *criteria event*: a different value function val or quality constraints $\mathcal{D}_{\mathrm{q}}$.
2. A *parameter event*: a change in the vector of uncontrollables $\bar{u}$ that does not require changes in the routing tree.
3. A *topology event*: a tree-link breaks, a node moves, a new node enters, or a node dies or leaves. This event includes moves of the sink. Contrary to parameter events, this would typically require an update of the routing tree. This event implies changes in $\bar{u}$ (e.g. the location) as well as potentially in $\mathcal{S}_{\mathrm{Pc}}$ (in case of a change in the set of nodes).

Criteria events are easy to handle, as our QoS optimiser computes and stores all Pareto points of the WSN, and these do not change. Hence, we only need to apply the new constraints and value function to the Pareto set $\mathcal{C}_{\mathrm{opt}}$ (perform the selection phase):

$$\bar{c}^* = \min(val(\mathcal{C}_{\mathrm{opt}} \cap \mathcal{D}_{\mathrm{q}})).$$

If the selected configuration is different from the current one, we need to load the controllable parameters of the new configuration $\bar{c}^*$ into the network (the loading phase).

Parameter events deal with a change in the vector of uncontrollables, say from $\bar{u}_0$ to $\bar{u}_1$. Examples are a change in the contention loss L or the transmission delay D_{tx} at some node. It is now likely that the configurations in $\mathcal{C}_{\mathrm{opt}}$ experience changes, at least in the metrics, but possibly also in the parameters (different parameter vectors are Pareto optimal). It is therefore necessary to recompute the Pareto set, and then do selection and loading.

For topology events, we specifically examine broken links and entering/leaving nodes, and implement a moving node as a node that leaves and comes back at a different location. We do study a mobile sink as a special case, because of its practical relevance and high impact. Other mobility scenarios are left as future work.

Parameter and topology events should be detected before they can be adapted to. For this to work, we need to rely on ways to measure these parameters on the nodes. It is possible, for example, to assess the value of the contention-loss parameter L from the models of Section 3.2, by maintaining a count on the packets that have collided versus the number of sent packets. It is also straightforward to measure the transmission-delay parameter D_{tx} and keep it as a running

average over a past period. If such assessment is not possible for certain parameters, there is no opportunity to react. Another interesting application of this is the ability to adapt to model inaccuracies. If uncontrollable parameters are seen as model constants that are estimated for the initial QoS-analysis phases, the real value of such parameters may be measured at run time, and the configuration can be adapted to using the more accurate model that has been obtained.

Topology events centre around changed link conditions. For an existing node to detect that its parent link is broken or too bad to be used, the parent and child could periodically reconfirm the link by a handshake, or use acknowledgements on the data messages from child to parent.

Reconfiguration obviously comes at a cost. The second objective in Section 3.3 is to minimise the cost of (re)configuration in terms of time and energy. It follows that there is a trade-off between the amount of time and energy spent on reconfiguration and the quality achieved by the task. This trade-off has two extremes. The first option achieves the best quality against the highest cost, by completely reconfiguring the network, which involves computation and/or communication at every node[1]. The other extreme is to do as little as possible to ensure that the task is still able to operate within the constraints, and nothing to improve the task quality beyond that. The latter would involve computation and communication at a minimum number of nodes. Between these two extremes lies a range of possible solutions, each with its own quality/cost trade-off. Section 4.4 discusses ways to set this trade-off. In this chapter, another control targeted at this trade-off is introduced: *locality*.

Definition 6.2 (Locality, Configured Area, (Minimal) Active Area). We define the *locality* of the (re)configuration process as the group of network nodes that play a role in the process, typically an interconnected group of nodes around the place where the event occurred. There is a part of the locality, in a region we term the *configured area*, of nodes that adapt their configuration after the event. These nodes perform at least the loading phase. We further consider a part of the locality, called the *active area*, of nodes that (apart from the loading phase) also play a role in the tree construction, QoS optimisation, and selection phases of the configuration process. The *minimal active area* comprises the nodes that necessarily need to be updated after an event (especially a topology event) to ensure that the network remains operational.

[1] Note that we assumed in Section 3.3 that task quality and configuration cost are independent. This basically implies that the frequency or granularity of reconfigurations is small enough, such that the task quality is not significantly affected by the energy and time used for reconfigurations.

Definition 6.3 (Deviation). The *deviation dev* of the reconfiguration process controls the size of the active area beyond the minimal active area. The active area is equal to the minimal active area plus nodes at most *dev* hops away from it.

Full task-quality optimisation without being concerned with the cost implies global configuration in general: the configured area is the whole network. Local reconfiguration generally does not ensure optimal quality. However, the active area does not need to be large. Reacting to a criteria event only requires selection and loading, and thus the active area is only the sink, even for optimal reconfiguration. As we see in this chapter, also reacting to the more comprehensive parameter or topology events usually does not require a large active area. In order to make the reconfiguration process local, the configured area is shrunk, such that nodes outside the area keep their parameters configured as they are. We typically make the configured area equal to the active area in these cases.

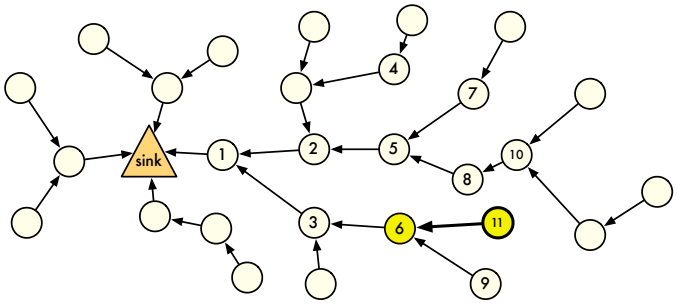
By means of the deviation parameter, the locality can be set to a wide range of sizes between the minimal active area and the full network. The deviation is therefore a powerful quality/cost trade-off control. As non-optimal (but cheaper) reconfiguration may deteriorate the quality over time, it is advisable to periodically reset the configuration by doing a global reconfiguration.

As in the previous chapters, reconfiguration can be done in either a centralised or a distributed manner, although local reconfiguration is typically a distributed affair. Especially adaptation of the routing tree is efficiently done in a distributed way. In this chapter, we focus on a fully distributed implementation, and clarify what changes when things are done centrally where applicable.

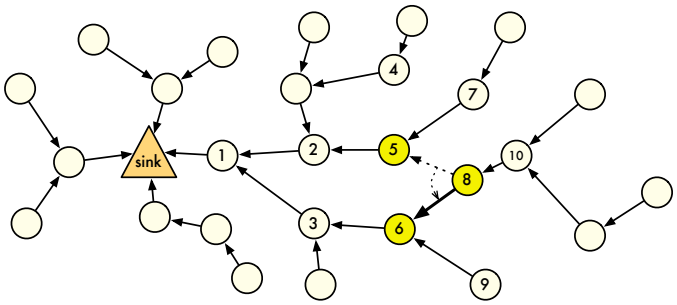
6.2 Basic Tree Maintenance

The occurrence of a topology event requires maintenance of the routing tree to be done before the other node parameters can be optimised. This section discusses how to adapt the tree after such an event. We assume this is done in a distributed way. Adaptation updates the existing routing tree, while striving to maintain the node degrees at a given value Δ or less, and minimising the paths lengths within this constraint, or by enforcing a shortest-path constraint (as in Chapter 5). We also consider a deviation parameter *dev* that restricts the locality in which the algorithm is allowed to act.

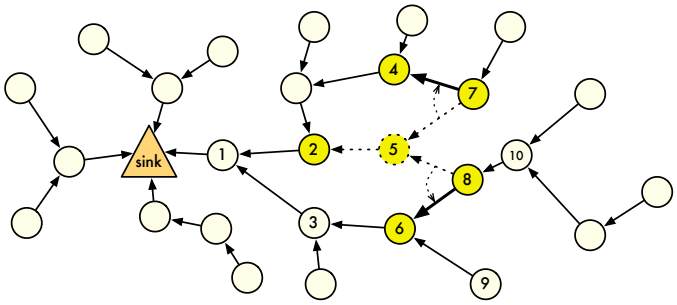
We discuss four types of topology events:



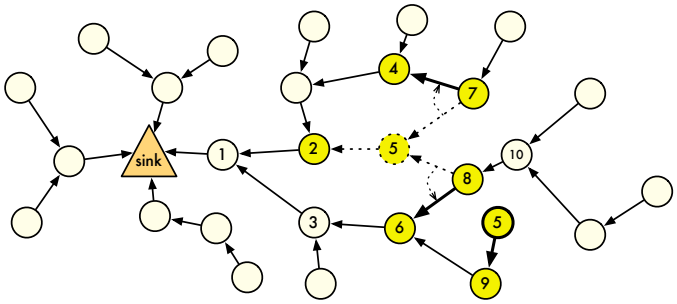
(a) Node 11 joins. It finds a parent in node 6.



(b) The link from node 8 to its parent node 5 breaks. Node 8 finds a new parent: node 6.



(c) Node 5 leaves. Nodes 7 and 8 are orphaned, and find new parents 4 and 6 respectively.



(d) Node 5 moves. The orphaned nodes 7 and 8 find new parents as in (c), and node 5 connects to node 9.

Figure 6.1: The four types of topology events. Thick arrows between nodes represent new links, dashed ones are broken links. Nodes in the minimal active area have a darker shade.

1. **Join.** A new node enters and needs to be incorporated in the running network. See Figure 6.1(a).

2. **Link breakage.** A tree-link becomes structurally so weak due to a change in the environment, that it cannot be used anymore (the reliability falls below a threshold). The nodes that used this link need to be rerouted. See Figure 6.1(b).

3. **Leave.** A node runs out of battery, breaks, or is taken away. The descendants of this node need to be reconnected to the sink. See Figure 6.1(c).

4. **Move.** A node moves to a different location. See Figure 6.1(d).

We only support occasional moves of sensor nodes. The underlying assumption is that sensor nodes are intended to be stationary, but may incidentally be moved by people or some effect in the node's surroundings. These moves are modelled as a leave followed by a join and are handled accordingly, as in Figure 6.1(d). In the next section, we study a mobile sink as a special case, because of its practical relevance and high impact. The simple leave/join mechanism does not work for a mobile sink. Other mobility scenarios are left as future work.

One might wonder whether the lifetime metric of Section 3.2 – defined as the time until the first node dies – still makes sense when nodes may die without ending the life of the network. We would like to re-emphasise that the definition of this metric intends to push the optimiser to balance the energy usage across all nodes. Ideally, all nodes would die at the same time. In practise, however, there will always be nodes that break earlier, while the network may still function properly. We simply treat this as a new instance of the configuration problem, in which the same metrics apply.

Note that joins, link breakages, and leaves have an important common property: one or more nodes need to find a new parent. A new node has never had a parent, a node that observes that the link to its parent is broken needs a new parent, and a node that disappears orphans its children. Therefore, the three cases can all be mapped to instances of a single problem, being the problem of (re)connecting a node to the tree. The minimal active area for a topology event contains all nodes that necessarily experience a change in child or parent nodes. All dark-shaded nodes in the four examples of Figure 6.1 belong to the minimal active area.

A node in search of a new parent acts similarly as a node that receives a Reduce Request message from its parent in the distributed degree-reduction algorithm of Section 5.4. The node will first broadcast a Parent Request message. This message contains the hop count (distance to the

sink) of the node's previous parent, or the value infinity for new nodes, and it has the value infinity in the degree field (note that in Section 5.4, the hop count and degree fields are values of the *parent* of the node that broadcasts a Parent Request). Each node receiving this request decides based on the same rules as in Section 5.4 whether it is a suitable parent. If so, it would answer with a Parent Reply message containing its hop count and node degree values. The requesting node assesses the replies and selects a parent from the candidate list using the function `ChooseParent` from Algorithm 5.2, given the degree target Δ. The choice of parent is confirmed to the new parent, which happens via a message called Child Confirm. This message should be acknowledged by the parent, and retransmitted if needed, to ensure the parent has a correct list of children. The correctness of this approach is ensured by the same reasoning as in Section 5.4.

In case no parents within the degree target were available (the new parent has a degree larger than Δ after the join), there may still be room for improvement. The parent may execute the degree reduction algorithm of Section 5.4, hoping that any of its other children is able to find another parent. Allowing this may cause a chain of reduce requests. We bound this chain to an active area with a range of *dev* hops around the node at which the first event occurred; outside the active area, nodes will not react to requests (*dev* is included in the messages, and decremented at each hop).

In order not to loose data, a node that is orphaned may store incoming data messages, to be forwarded to the new parent upon connection. Alternatively, data loss may be accepted. We refer to the time between losing connection and reconnecting to a new parent as the *disruption time*, which is one of the evaluation metrics in the simulations of Section 6.5.

Multiple tree events that occur simultaneously may require repairing of the tree at various places at the same time. This is possible for the same reasons as for the degree reduction algorithm: nodes that are busy reacting to one event, will ignore messages from other events.

6.3 Tree Maintenance for a Mobile Sink

Supporting a mobile sink is of interest for lifetime extension, as it relieves the energy bottleneck that naturally exists at nodes close to the sink, which need to transfer much more data than nodes further away [39, 65]. Furthermore, the application may have the need for a mobile sink, for example in disaster-recovery situations in which rescue workers walk around with handheld devices to collect information about the scene.

As the routing tree most likely breaks when the sink moves, the tree needs to be reconstructed

(see Figure 6.2(a)). We assume that the sink moves stepwise, and after each step resides at its position long enough to justify rebuilding the tree. For applications with a continuously and relatively fast moving sink, maintaining a routing tree is probably not the best solution and other methods of delivering data to the sink may be more suitable [17].

As before, our goal is to create a tree in which all nodes have a degree no more than the degree target Δ and paths that are as short as possible within that constraint, or an SPST with degrees as low as possible (depending on the chosen meta trade-off, see Chapter 5). We assume that such a tree gives rise to the best task quality. After a move of the sink, the cost of globally reconstructing the tree (in time and energy) may be too high, especially if moves are frequent. We therefore propose a new algorithm that recreates the tree only in a certain region around the sink, the active area, and retains the parts of the existing tree elsewhere, thereby sacrificing some quality (longer paths). The size of the active area is determined by the deviation parameter *dev* as before.

6.3.1 Minimal-Cost Reconstruction

We consider full tree reconfiguration as in Chapter 5 as a baseline algorithm that provides the best quality against the highest cost. At the other end of the spectrum would be an algorithm that has the lowest reconfiguration cost, but a lower quality as well. This algorithm ensures that all nodes are connected to the sink again, and is therefore required for a minimum service level, but does nothing beyond that to improve the quality. The active area for this algorithm is therefore considered to be the minimal active area. We call this algorithm *QuickFix*. We first introduce QuickFix, and then explain under which assumptions it works properly and which further measures may be needed.

After the sink moves to a new position, it may be out of range of some or all of its children in the existing tree. QuickFix creates new paths from these nodes to the sink. All other nodes in the network are descendants of the former children of the sink. Therefore, reconnecting these former children to the sink means that all nodes are connected again. Reconnection is based on the observation that the existing tree has paths to the sink's former child nodes from anywhere in the area, and happens in three phases; see Figure 6.2 for an overview. The sink, at its new position, starts by broadcasting a SinkMove message (see Table 6.1 for the full format of this message). The nodes that receive this message become the sink's new children, which all have a path to one of the disconnected former children of the sink. In the second phase, QuickFix follows these paths

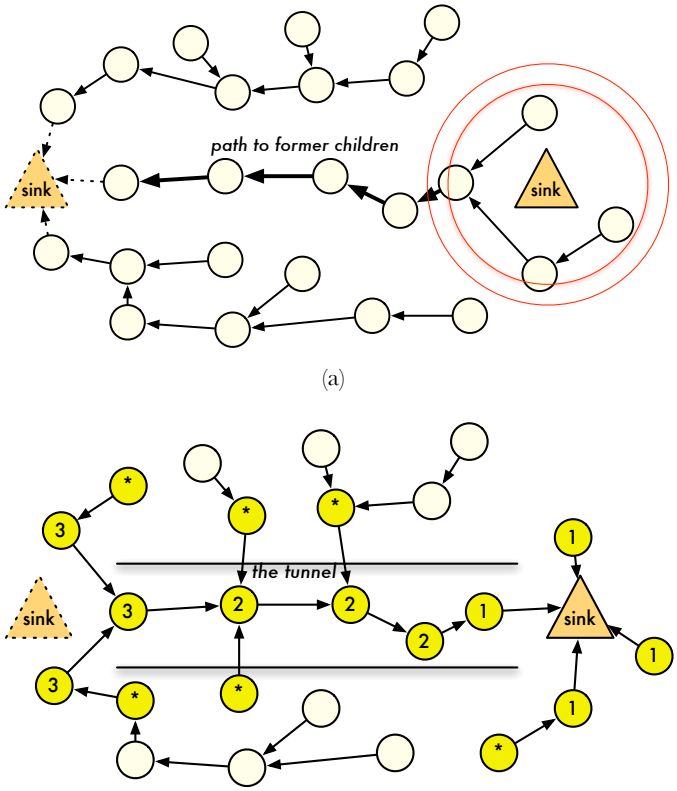


Figure 6.2: The sink moved from left (dashed triangle) to right. (a) The nodes that have dashed arrows became disconnected after the move of the sink. Thick arrows indicate a path from the new position of the sink to these nodes. (b) Dark-coloured nodes form the active area of QuickFix. The number indicates the QuickFix phase they perform. Nodes marked with an asterisk (*) are the nodes just outside the tunnel that connect as well, and may start Controlled Flooding.

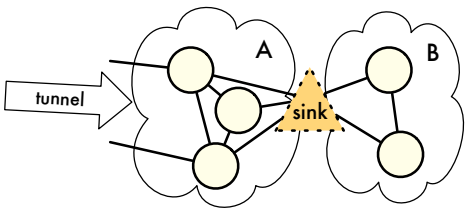


Figure 6.3: Former children of the sink, in two disconnected groups. Group A is connected to the tunnel, but none of these nodes can reach a node in group B by broadcasting. Group B's nodes (plus all descendants) remain disconnected after QuickFix.

Table 6.1: SinkMove-message format

Field	Explanation
Sender	ID of the sender node
Destination	ID of the destination node, or ∅ if broadcast
Deviation	Deviation value *dev*
Hop count	Number of hops to the sink in the new tree
Update number	Number of the current maintenance pass; incremented at each move

and reverses the links, until such a former child is reached. To this end, each node involved in this phase sends a unicast SinkMove message to its current parent upon receiving a SinkMove message from a child or the sink, after which the sender of the SinkMove message is made the new parent. Finally, this former child node broadcasts a SinkMove message that enables other former children of the sink to connect, which is repeated until all are reconnected.

A state diagram for the QuickFix algorithm is shown in Figure 6.4. The initial *adaptation* phase in this diagram is entered immediately after the loading phase completes (see Figure 4.6). Each phase of QuickFix has its own state, which is reached by a node after receiving a SinkMove message that matches a certain condition (indicated at the transition's arrow). To ensure that nodes react to a SinkMove message only once, an update number is used in the SinkMove-message format. This number is incremented at each reconfiguration (sink move), and only if a node receives a SinkMove message which has a higher update number than it has seen before, it will update its parent variable and forward the message. A node stays in one of the QuickFix states for a duration specified by a SinkMove timer (for reasons described in the next sub-section), before performing its actions and proceeding to the *degree reduction* phase. Degree reduction happens as in Chapter 5.

QuickFix effectively creates a *tunnel*, containing all above mentioned paths, through which all disconnected nodes are reconnected to the sink. This has no effect on the node degrees (except the sink's), but all paths are enlarged by paths of the tunnel. An optimisation that does not cost any extra SinkMove transmissions can be made: any node that overhears a SinkMove message may set the sender of this message as its parent node (nodes marked by an asterisk in Figure 6.2(b)). By doing this, the node creates a shortcut to the tunnel and shortens the path of itself and its descendants.

Proposition 6.1 (QuickFix). *If a tree exists before the move of the sink, QuickFix leads to a tree containing all nodes and the sink (at its new position) as root, under the following conditions:*

1. *The sink's broadcast after the move is received by at least one sensor node.*
2. *QuickFix messages creating the tunnel are not lost.*
3. *The sub-network consisting of only the former children of the sink is fully connected (see Figure 6.3).*

Proof. In the current tree T_0, all nodes have a path to one of the sink's former immediate child nodes, a set called S_0. After the sink's move, it starts a new tree T_1 by connecting a number of sensor nodes within its range, as set S_1 (phase 1). Condition 1 ensures that this is possible.

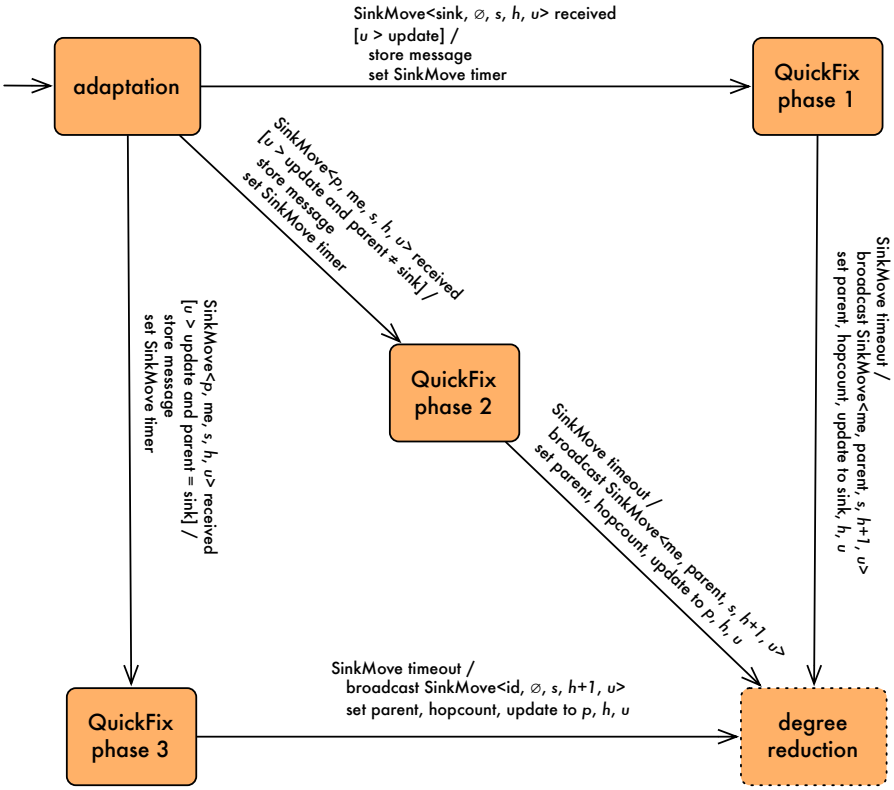


Figure 6.4: QuickFix, state diagram. This diagram follows immediately after Figure 4.6; the initial adaptation *state is reached after completing the loading phase. State transitions are triggered by events and/or conditions as annotated at the arrow before the slash. An event can be due to an incoming message from another node or a timer expiry. Actions at a transition are given after the slash. All events that are not listed at a state are ignored. The degree-reduction state is the entry point for the degree-reduction algorithm of Section 5.4. The values of the fields of SinkMove messages are given between angular brackets in the order as in Table 6.1. The three phases of QuickFix each have their own state and are triggered by a SinkMove message in slightly different conditions.*

Further, each node in S_1 has a path to a node in S_0 in T_0. Phase 2 grows tree T_1 by reversing the T_0 links on the paths from S_1 nodes to S_0 nodes. The nodes on these paths form the tunnel. Condition 2 ensures the reliability of these paths. Finally, condition 3 ensures that the sub-set of nodes in S_0 that was reached in phase 2 will connect to the remaining S_0 nodes in phase 3. Consequently, all nodes in S_0, S_1 and the tunnel become part of T_1. As all other nodes in the network have paths in T_0 to a node in either S_0, S_1 or the tunnel, all nodes are now part of T_1.□

The first condition in Proposition 6.1 implies that the sink needs acknowledgements from its new children that have received its broadcast. If no acknowledgement is received, the sink rebroadcasts. The second condition can also be guaranteed by an acknowledgement scheme, as all transmissions are unicast. The third condition will generally be met if the node density (with respect to the radio range) is sufficiently high. This will usually be the case, as WSNs are typically very dense.

6.3.2 Improving the Quality

QuickFix reconnects all nodes to the sink in a highly cost-efficient way, but the average path length of the resulting tree will be high. The active area consists of only the old and new children of the sink and the tunnel between them. To reduce the average path length, but keep the costs limited, we use another mechanism on top of QuickFix, which enlarges the active area by a number of hops that is specified by the deviation parameter *dev*. This parameter is part of the SinkMove-message format (see Table 6.1). Any node that overhears a SinkMove message does not only connect to the sender (as described in the previous sub-section), but if the deviation value is larger than zero, it will broadcast a new SinkMove message with a decremented deviation value, and an empty destination field to indicate a broadcast. We refer to this as *Controlled Flooding* (CF). By flooding the active area, a local SPST is constructed, and consequently also the paths of the nodes outside the area are reduced in length (see Figure 6.5 and the state diagram in Figure 6.6).

QuickFix and CF are not executed consecutively, but run in parallel; SinkMove messages for the three phases of QuickFix as well as CF are distinguished by different conditions (see the transitions in the state diagrams). The use of update number in the SinkMove message format, which are incremented at each sink move, ensures that a node reacts to a move only once. There is one exception to this rule: since QuickFix is crucial to reconnect all nodes in the new tree, QuickFix's SinkMove messages are always forwarded (to the parent in the *old* tree!), even though

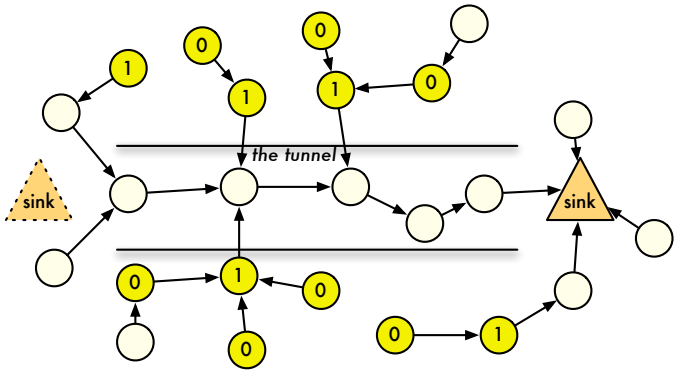


Figure 6.5: The example of Figure 6.2 with Controlled Flooding (CF) having $dev = 1$. *Dark-coloured nodes have been affected by CF and some of them (those in the bottom part) have found a shortcut to the sink. The numbers indicate the deviation values* dev *per node.*

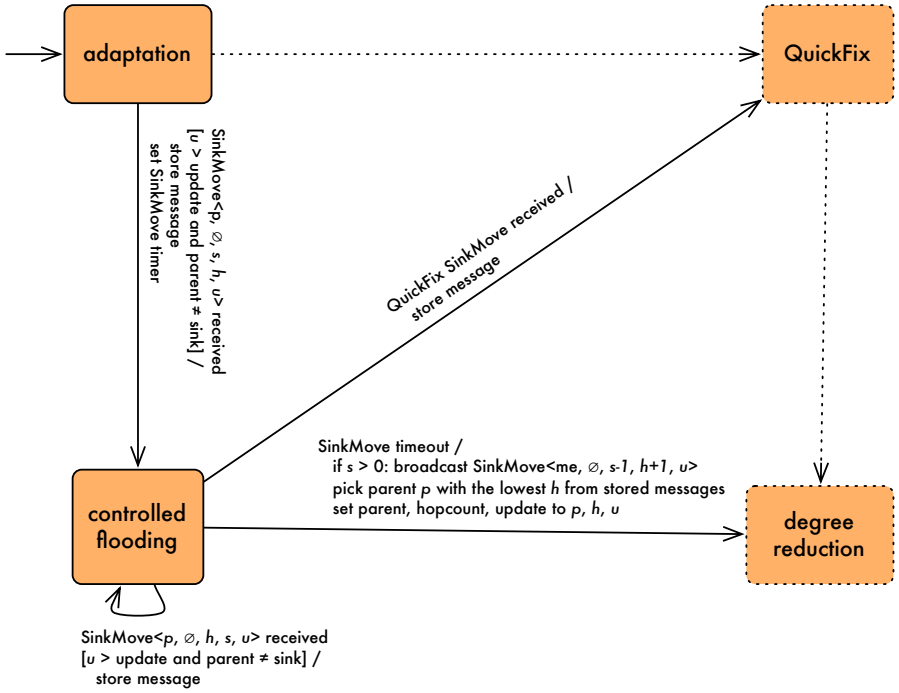


Figure 6.6: Controlled Flooding, state diagram. This diagram extends the QuickFix diagram of Figure 6.4; the three QuickFix states are represented by a single QuickFix state in this diagram. The controlled-flooding state is also triggered by a SinkMove message. A CF message is distinguished from a QuickFix message by the different conditions on the transition edges. In the controlled flooding state, new SinkMove messages may arrive. If such a message meets any of the QuickFix phases' conditions, CF is aborted and QuickFix is executed as in Figure 6.4.

a CF message with the same update number arrived earlier.

Before reacting to a Flood message, the distributed tree-construction algorithm of Section 5.4 waits for a short while to collect potential Flood messages from other nodes, in order to mitigate the differences in propagation speed of the messages. The sender of the Flood message with the shortest hop count is chosen as the new parent, and then the Flood message is forwarded. Controlled Flooding does the same. To ensure that QuickFix and CF's SinkMove messages travel through the network with about the same speed, just like Flood messages, also QuickFix uses this delay. Both delays are implemented in the state diagrams via the SinkMove timer.

Since all chains of forwarded SinkMove messages (QuickFix and CF) originate from the sink, each affected link is pointed to the node that sent the message, and nodes react only once to CF messages of a certain update number, a correct tree (rooted at the sink, loop-free) is formed in the active area. The nodes at the edge of the active area keep their existing sub-trees, so all nodes are connected to the active area and hence to the sink. Loss of CF messages may lead to longer paths, but never results in a broken tree, as QuickFix already takes care of connecting all nodes.

After QuickFix and CF finish, the node degrees are reduced as before. However, paths have changed in length and only the nodes in the active area know their distance to the sink. Therefore, only nodes in the active area take part in the degree reduction in order to guarantee loop-freeness (the improvement rules in Section 5.4 need the hop count). When only QuickFix is used, the reduction algorithm is not able to do much, since the active area is small. Therefore, we bound the number of nodes that may directly connect to the sink by Δ, already in the first phase of QuickFix via some extra handshaking.

The deviation parameter controls the trade-off between reconfiguration cost and quality. A larger deviation value leads to a larger active area, and thus to more nodes obtaining shorter paths, and a better quality. On the other hand, reconfiguring a larger active area takes more time and more transmitted SinkMove messages. The best value for the deviation parameter depends on the application.

6.3.3 Related Work

Several topology-maintenance schemes have been suggested earlier, such as STEM [58], DTM [6], SSP [69], MobiRoute [40], and DCTC [72], some of which aim at mobile sinks or targets. In STEM, an additional transceiver is required for control messages. Such a method may increase the size and cost of micro-sensors. DTM, on the other hand, constructs an optimal tree as

the mobile target moves. Knowledge of the movement pattern of the mobile target, however, is required in DTM. Although such knowledge is not required in SSP, this protocol requires flooding the network with control messages twice on every sink update. This introduces a high communication overhead, as well as numerous changes to the topology. Our method, similar to SSP, does not assume knowledge of the sink movement pattern or localisation. However, we take a further step over SSP to restrict the amount of topological changes and control messages, and to balance out energy consumption among nodes in the network.

Luo and Hubaux [39] assume that data packets are generally geographically routed towards the mobile sink and conclude that a sink circling around the perimeter of the network is beneficial for the network's lifetime. In MobiRoute, the same authors suggest a routing mechanism to support this concept, in which they assume the sink's trajectory adaptively controlled to maximise lifetime. Topological changes are propagated throughout the whole network when the sink reaches a new anchor point, which is expensive.

Kim et al. [32] provide a solution for a scenario with multiple moving sinks and a single data source, in which they create and maintain a dissemination tree rooted at the source. This is in contrast with our case, with a single sink and multiple sources (all sensor nodes are sources). Moreover, such a dissemination tree does not span all nodes.

Akkaya and Younis [3] in EARM also identify a trade-off between maintenance costs and efficiency of the topology. When the sink moves, they first try to increase the transmission range of the last-hop nodes to maintain connections with the sink. If this is no longer possible, intermediate nodes are found and added to the routes. Only if both options fail or the topology becomes too inefficient, the whole network is reconfigured, which causes a lot of overhead. While this may be sufficient for a relatively slow-moving sink, this mechanism would still need complete reconfigurations quite often when the sink travels faster. Our method, on the other hand, allows for better fine-tuning of the trade-off, such that complete reconfigurations are much less needed.

The tree-reconstruction method of Zhang and Cao [72] (DCTC) comes closest to our method, as they also flood a restricted area. However, our way of combining such restricted flooding with a baseline mechanism that ensures connectivity is new. By doing this, we enable a wide range of possible trade-offs between maintenance costs and task-level quality metrics.

QuickFix is similar to the Arrow Distributed Directory Protocol introduced in a different context by Demmer and Herlihy [20]. The Arrow protocol also maintains a spanning tree on a network graph, but the situation is slightly different, as there is no sink node that actually moves

around. Instead, the root of the tree is a node as any other, but other nodes in the network can request to become the new root. This request happens as in QuickFix's second phase: a control message is sent on the path from the requesting node to the root, while all links on the path are reversed. The difference with the mobile sink case is therefore that the old and new root are two different, static nodes and the network graph does not change. The old and new root are still/already part of the tree after the change event, and none of the nodes becomes disconnected. Hence, the first and third phases of QuickFix are not part of the Arrow protocol.

Contrary to many existing approaches, our tree-reconstruction method does not require any knowledge about the deployment of nodes or movement of the sink, and is robust to message loss. Moreover, our way of integrating topology control with node configuration to meet task-level QoS goals, as detailed in the next section, is unique.

6.4 Optimising Node Parameters

Normal operation of the network task can continue as soon as the tree has been reconstructed. However, due to the changes in the structure of the network, the level of quality achieved by the running task is typically lower than possible, and could even be such that QoS constraints are violated. Furthermore, uncontrollable parameters may change over time (a parameter event), also changing the WSN configuration. It is therefore worthwhile to improve the quality by reconfiguring the nodes' parameters after a change in the network.

Parameter and topology events occur at a certain location in the network. When a node changes its parent due to any of the topology events of Section 6.2, the node itself, but also its old and new parent play an active role: the minimal active area of the reconfiguration process comprises these three nodes (see Figure 6.1). The size of the active area also can be larger, depending on the deviation parameter. In the mobile-sink case, many nodes change their parents, and the active area is always a region around the sink. Uncontrollable parameters are present at the lowest level: the node level. Suppose a single value in $\bar{u}$ changes, then this is the value of an uncontrollable parameter belonging to a single node. The minimal active area contains only this node. We show in this section that the active area for parameter optimisation contains at least the areas described above, and potentially more nodes if global optimality is required. An overview of the various types of parameter reconfiguration with the associated active and configured areas is given in Table 6.2. The meaning of each row in this table becomes clear in this section.

While parameter reconfiguration is in progress, the network is in a state of reduced quality. It

Table 6.2: Types of parameter reconfiguration with varying localities

Type	Active area	Configured area
Naive global	Whole network	Whole network
Efficient global	Nodes near event given dev; paths to sink	Whole network
Semi local	Nodes near event given dev, paths to sink	Nodes near event given dev
Fully local	Nodes near event given dev	Same as active area

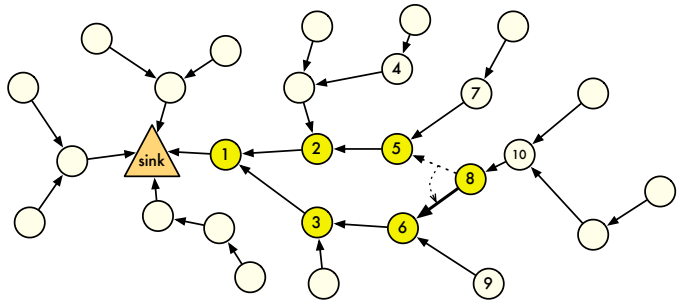


Figure 6.7: Node 8 has changed from node 5 to node 6 as its parent. The minimal active area comprises these three nodes. All dark-shaded nodes and the sink need to recompute their Pareto points and form the active area for a globally-optimal reconfiguration.

is therefore desirable to reconfigure as quickly as possible. Moreover, parameter reconfiguration comes at a cost, as processing and communication is needed to compute and load the new settings. In this section, we first look at how to efficiently compute a globally-optimal post-event configuration. We then explore the trade-off between the quality achieved by reconfiguration and the cost it has by introducing localised optimisation strategies.

6.4.1 Globally-Optimal Reconfiguration

In this section, we assume that the location and parent of each node do not change anymore. Hence, the configuration space $\mathcal{S}_{\text{Pc}}$ reduces to $\mathcal{S}_{\text{Pc}}|_T$ given the current tree T, and the reconfiguration phases needed are QoS optimisation, selection and loading. Furthermore, after initially configuring the network and after each reconfiguration, all cluster-level Pareto sets and the indexing tables are stored and available for reuse.

The most straightforward, though inefficient, way to globally optimise the QoS is to simply re-run the full QoS-optimisation algorithm from Chapter 4, on the whole network. Observe however, that as events occur locally, many nodes and their sub-trees/clusters remain unchanged. Therefore, also the sets of Pareto-optimal configurations for many nodes and clusters outside the minimal active area do not change, and need not be recomputed. In Table 6.2, the former is called *naive global* parameter reconfiguration, and the latter is termed *efficient global* parameter

reconfiguration.

Suppose the uncontrollables of a node i change in a fully configured network. As a result, only the clusters containing node i could have changed Pareto sets. Due to the clustering order of Algorithm 4.4 in which all created clusters are leaf clusters (as in Definition 4.3), the clusters containing i are the ones having as roots all nodes on the path from i to the sink (including node i). Therefore, node i and all its ascendants form the active area for global adaptation to this event. In Figure 6.7, for example, when a parameter event occurs at node 6, this node recomputes the Pareto set for its cluster, using the Pareto sets of its children, which have been stored since the previous configuration process ended. The new Pareto set is passed to node 3, which uses this plus its current one-node cluster set and the Pareto sets of its other children to recompute its cluster-level Pareto set. This is repeated at node 1 and the sink, after which the sink performs the selection phase and initiates the loading of the new configuration.

Now consider a fully configured network in which a single node changes its parent as part of the reaction to a topology event (including the mobile sink case), as in Figure 6.7, where node 8 switches from node 5 to 6. This would cause changes in the Pareto set of the cluster with root node 8. Further, the roots of all other clusters that have changes in them need to be updated: the clusters with as root the old and new parent (nodes 5 and 6), and all nodes on the paths from these three nodes to the sink (nodes 1, 2 and 3). These six nodes form the active area for global adaptation to this event.

The clustering order implies that for a globally-optimal parameter reconfiguration, the QoS-optimisation algorithm may start at the nodes at the edge of the active area, further referred to as the *boundary nodes*, instead of at the leaf nodes of the network.

Definition 6.4 (Boundary node). A boundary node of an active area is a node that has no descendants inside this active area.

In Figure 6.7, after the switch, nodes 5 and 8 are the boundary nodes, and for a parameter event at node i, node i is the (only) boundary node. The reconfiguration of the active area reuses the Pareto sets of the clusters just outside the area. Note, however, that the newly selected configuration at the sink, may cause a different configuration to be selected from the Pareto sets of *any node*, including the nodes outside the active area. This means that, while recomputing the Pareto sets is local, loading the selected configuration still involves all nodes in the network, and thus the configured area is the whole network.

6.4.2 Localised Reconfiguration

To make the reconfiguration completely local, not only the tree reconstruction and QoS analysis phases, but also the loading phase should be restricted to a local area. In fact, after the tree has been reconfigured, the network is already able to operate, and therefore the lowest-cost action is simply to do nothing at all, and keep the current configurations in all nodes. However, we would still need to ensure that the constraints are satisfied, so parameter reconfiguration may still be needed.

Low-Cost Adaptation to Parameter Events. In case of a changed uncontrollable at node i, the most localised reconfiguration option is to recompute only the Pareto set of node i's cluster, without propagating the results to i's ascendants. For local reconfiguration, we do not wish to adapt the configurations of i's descendants, and therefore we use only the current configurations of i's children in the computation (instead of the full Pareto sets). This has the added benefit that the analysis becomes simpler (smaller configuration space), significantly reducing the cost of reconfiguration. The price is sub-optimality of the found task-level Pareto set and hence a potentially non-optimal quality of the selected configuration.

We write $\bar{c}_i^*$ for the currently selected configuration of node i. The one-node cluster Pareto set for node i given the new vector $\bar{u}_1$ is computed. Then, the cluster-level Pareto set $\mathcal{C}_i|_{\bar{u}_1}$ is computed from the new one-node cluster Pareto set and the current configurations of i's child clusters. The issue is that quality constraints (as well as the value function) are defined only at the task level, so we cannot use lower cluster-level configurations to draw conclusions about quality-constraint satisfaction. What we do know, is that the current configuration satisfies the constraints. This implies that each new configuration that dominates the current configuration, also satisfies the task-level constraints owing to the monotonicity of the mapping functions (see Chapter 4), which enables us to conservatively pick a new configuration: if the set

$$\mathcal{C}^* = \{\bar{c} \mid \bar{c} \preceq \bar{c}_i^*, \bar{c} \in \mathcal{C}_i|_{\bar{u}_1}\} \tag{6.1}$$

contains at least one configuration, we may use it. If $\mathcal{C}^*$ is empty, we do not know which of the configurations satisfies the task-level quality constraints. We could then select the configuration that is nearest to $\bar{c}_i^*$, for instance in terms of (2.4), and hope the impact on the task level quality metrics is small or insignificant. If this is not acceptable, we could propagate the new Pareto

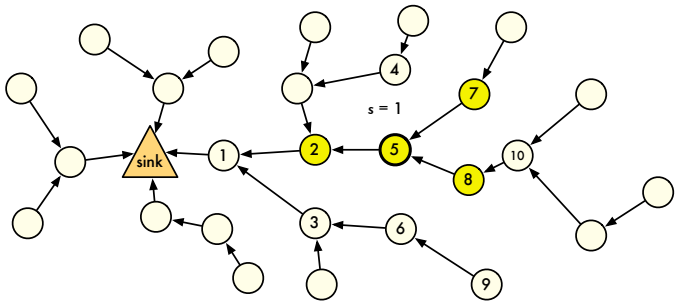


Figure 6.8: Node 5 experiences a change in its uncontrollable parameters. The dark-shaded nodes form the active area for the local parameter reconfiguration to this event with $dev = 1$. *Nodes 7 and 8 are the boundary nodes.*

set to the sink to derive the task-level metrics and determine the optimal constraint-satisfying configuration. The active area is then enlarged (i plus all its ascendants), while the configured area is still only node i. This is similar to global reconfiguration as in the previous section (the active area is the same), but the configured area is just one node instead of the whole network. We refer to the latter approach as *semi-local* parameter reconfiguration, in contrast to *fully-local* (see Table 6.2).

Semi-local reconfiguration also ensures that all of i's ascendants have correct Pareto sets. After a fully-local reconfiguration at node i, the Pareto sets of the ascendants of i are not accurate anymore. However, if i conservatively picks a new configuration as above, monotonicity ensures that also the current configurations of i's ascendants contain conservative estimates of the metrics. Hence, if the current configurations of these ascendants are used in later local reconfigurations, the results would also be conservative.

Improved Quality for Parameter Events. If the low-cost reconfiguration described above is not good enough, the optimiser can be given more freedom, by reconfiguring more nodes around i: some of its descendants and ascendants. Intuitively, it makes sense to update the configurations of these nearby nodes, as i's performance is influenced most by its neighbourhood. The size of both the active and configured area around i is controlled by the deviation parameter dev: the areas comprise i plus its ascendants and descendants at most dev hops away. This is illustrated in Figure 6.8. The earlier case in which only i is reconfigured has a deviation value $dev = 0$. The active area is a sub-tree of the network's routing tree (a cluster according to Definition 3.1; not necessarily a leaf cluster), while all descendants of its boundary nodes have already been configured. We can therefore apply an adapted form of the algorithm of the QoS-optimiser of Chapter 4 to compute the new Pareto points.

As we only allow nodes in the configured area to update their configurations, the QoS optimiser needs to take into account the current configuration for all other nodes. Moreover, the optimiser needs to obey the leaf-to-root clustering order as before. Therefore, as for global reconfiguration, the process has to start at the boundary nodes, and these nodes use the current configurations of the clusters just outside the area (see Figure 6.8). Note that this is a generalisation of the case with $dev = 0$ above, in which i is also a boundary node.

When the optimiser has computed the Pareto set of the cluster of the active area's root node, which is the cluster that contains the whole configured area, we need to determine which configurations meet the quality constraints. As above, the only certainty we have at this (cluster) level, is that all configurations that dominate the current configuration meet the constraints. Hence, we again apply (6.1) to find these points. If there are no such points, we could either choose the nearest to the current configuration, or extend the active area by continuing the configuration process up to the sink, in order to compute the task-level configurations and apply the constraints and value function on these (semi-local parameter reconfiguration). A small step further is to make all nodes on the path to the sink part of the configured area, and compute new parameters for these as well.

Topology Events. For local adaptation to topology events, we assume that all nodes in the active area for tree reconstruction also undergo parameter optimisation. We also equate the configured area to the active area. To further exchange quality for lower cost, we could reduce the configured area even more (smaller than the area of tree reconfiguration). However, not re-analysing all clusters with changed parents or children inside means that the computed metrics are not accurate; when locally reconfiguring the whole active area from boundary nodes to the local roots, the computed metrics are always accurate.

In case of a topology event, multiple nodes may be changed together. Especially in the mobile sink scenario, the affected area may be a relatively large region. It makes sense to reconfigure the parameters of nodes that form a cluster (sub-tree) in the affected area together. Each cluster can then be reconfigured locally, in the same way as for the parameter event above. As for parameter events, both semi- and fully-local reconfiguration are possible, with the same pros and cons. In case of semi-local reconfiguration, the affected area is always a single cluster. The recomputed configurations are then task-level configurations and can therefore be directly checked for constraint satisfaction and value. As the active (and configured) area in the mobile sink case

is always a region around the sink, we only consider semi-local reconfiguration, for the added benefits it has against low additional costs compared to fully-local in this specific case.

6.4.3 Practical Details

Finding the Active Area and Boundary Nodes. From the previous, it follows that we need to make the boundary nodes start the QoS analysis with the correct child Pareto sets. However, after a topology event, a node actually does not know whether it is a boundary node or not. What is more, in case of global and semi-local reconfiguration, not every node may know that it is a part of the active area. In the example of Figure 6.7 in which node 8 changed parents, only nodes 5 and 6 will be aware of the change, while also 1, 2, and 3 need to be updated.

In these cases, we therefore make every changed node send a message called Optimise to its parent (after some delay to ensure the tree is stable), which indicates that the parent is part of the active area. This message needs to be communicated reliably (acknowledged, and retransmitted if needed). If the parent was not a changed node, it now knows that it is also in the active area, and it will forward the message to its own parent. When this procedure completes, all nodes in the active area are identified, and these nodes also know whether each of their children is in the active area or not.

A node is identified as a boundary node if it does not receive an Optimise message from any of its children within a reasonable period of time. The boundary nodes then start the QoS analysis, which propagates in the usual way up to the root. Each node that has one or more children outside the active area requests these children to transfer their current Pareto sets (these may have changed as a result of other events), or only the currently selected configuration in case of semi-local reconfiguration, before commencing the QoS analysis. After completing the QoS-optimisation phase, the sink proceeds with the loading phase. When the load messages reach outside the active area, they are no longer forwarded in the localised case. In the globally-optimal case, the load messages are forwarded up to the leaf nodes of the network.

In the fully-local adaptation to a topology event, we may restrict parameter optimisation to the nodes affected by the repairing of the tree. It is likely that there will be multiple separate clusters in the active area, which each should do their own optimisation. Boundary nodes are appointed through the mechanism described above, though without nodes that were not affected in the tree reconstruction playing a role. A cluster root is identified by its parent that is outside the active area.

For a fully-local adaptation to a parameter event at node i (as in Figure 6.8), the boundary nodes are found by simply descending the tree by dev hops. These boundary nodes know that they need to start the analysis and may do so straight away, and thus a procedure as above is not needed. Also the root of the active area – the node at dev hops towards the sink from i – knows that it is the root when the messages reach there, and that it should finish the configuration phase and perform the selection phase.

Until here, we assumed that all adaptation takes place in a distributed way. While this is certainly the most natural way to update the routing tree, centralised execution may still be the method of choice for parameter optimisation due to the potentially high computation costs on sensor nodes. Such centralised parameter optimisation would need all changes to be communicated to the sink, very much like during the initialisation phase of the configuration process as defined in Section 3.4. To save costs, new parameters may be computed and loaded periodically, instead of after each change.

Concurrent Events. If fully-local parameter optimisation takes place due to two events, in two clusters that are disjoint, no interference occurs and the clusters may safely be optimised simultaneously. If the clusters overlap, that is, the root of one cluster is part of the other cluster, one of the clusters needs to be optimised first. The second reconfiguration action is either aborted at the node that joins the two clusters, or suspended, and resumed after the first reconfiguration completes.

For semi-local and global reconfiguration, the active areas for multiple events always overlap, as the sink is always involved. Reconfiguration for multiple events is then always handled sequentially. Because reacting to each event may lead to a large overhead if many events occur, one could decide to do periodic global reconfigurations instead. Further details about these concurrency issues are left as future work.

6.5 Experiments

Since the performance of our reconfiguration approach depends on many factors, such as the type of task, the size of the network, the frequency and place of events – such as the movement pattern of the sink (size of steps, speed) – we use simulations to compare various scenarios. We are especially interested in the choice between localised and globalised QoS analysis on resulting task quality and reconfiguration costs. To see whether parameter reconfiguration really makes

sense, we also compare the results with the option of not reconfiguring at all (though minimal tree reconstruction may still be needed to ensure the network remains functioning).

6.5.1 Simulation Overview

The simulations were carried out in the same basic set-up as in the previous chapters; see Section 4.6 for details. We used networks of 900 TelosB sensor nodes randomly deployed in an area of 300×300 m, and running the target-tracking task defined in Section 3.2, and 27 different configurations per node. We only focus on the distributed execution of the (re)configuration algorithms. We distinguish five kinds of reconfiguration: the four of Table 6.2 plus no reconfiguration at all.

For local parameter reconfiguration, a value function *val* is needed that chooses one of the Pareto points to be loaded into the network – local reconfiguration depends on the *current* configuration, that is, one that was selected earlier as having the best value. For easy comparison of the various methods, we use a value function that assigns a real value to a configuration: a weighted sum of all four metrics, where each weight normalises the metric. To this end, we define the value of a configuration vector $\bar{c}$ = (information completeness, detection speed, lifetime, coverage degree) as its inner product with the vector $\bar{v} = (100, 200, 0.1, 100)$:

$$val(\bar{c}) = \sum_{i=0}^{|\bar{c}|-1} \bar{c}[i] \cdot \bar{v}[i] \tag{6.2}$$

We do not use constraints in these experiments, to avoid biased results.

We simulate three scenarios. Firstly, we see what happens when uncontrollable parameters change, and whether reconfiguration is useful. Secondly, we look at a scenario in which nodes break – due to a drained battery or some other defect – a situation that is very relevant in practise. Finally, we examine the mobile sink scenario. We do not simulate criteria events, since these only require the selecting and loading phase to be redone, exactly as in Chapter 4.

6.5.2 Changing Uncontrollable Parameters

The contention-loss factor L is one of the uncontrollable parameters in the node model of Section 3.2. It is a number in the range $[0, 1]$, which is an estimate of the fraction of the packets transmitted to the node's parent that are lost due to collisions. This is typically a parameter that is estimated at design time, but may turn out to be quite different when running the task on the network. In this experiment we do an initial configuration of the network in which all nodes have

$L = 0.1$, and store the set of Pareto points $\mathcal{C}_0$. Then we change the L values at all nodes to random values from the set $\{0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3\}$ (a parameter event), and recompute the metrics for the configurations in $\mathcal{C}_0$ (having the same controllable-parameter vectors that is); the resulting set is called $\mathcal{C}_1$. The metrics of the configurations in $\mathcal{C}_1$ correspond to the metrics of the system after the event if we would not reconfigure the parameters. After that, the network is globally reconfigured (efficient global), and the quality difference $D(\mathcal{C}_2, \mathcal{C}_1)$ between the new Pareto set $\mathcal{C}_2$ and $\mathcal{C}_1$ is determined (see Definition 2.4).

The simulations are done for 100 random networks, as in the previous chapters. The results are somewhat surprising: the quality difference between not reconfiguring ($\mathcal{C}_1$) and the real Pareto points after the event ($\mathcal{C}_2$) is negligible (not exactly zero due to quantisation differences, but always smaller than 0.001). This implies that the Pareto points that were initially computed – for the "wrong" values for uncontrollable parameter L – are the same as the Pareto points with corrected or changed L values. In other words, the set of Pareto points seems to be independent of the values of L of the nodes, at least in this model. That is, it seems that the same controllable-parameter vectors are Pareto-optimal for any L. This is understandable from Figure 3.3, in which L has a monotone effect on the reliability metric (the other metrics are not affected by L, since retransmissions are not used in our task models).

Other tests reveal that the same holds for the other uncontrollables in the model. The values of the *metrics*, however, do change. Therefore, in order to check the constraints, it is still needed to recompute the metrics, but only of the Pareto-optimal configurations in $\mathcal{C}_0$. This is obviously much more efficient than when starting with the full parameter space as in the original configuration problem. After the recomputation of the metrics, we re-execute the selection and load phases (load only if the newly selected configuration is different from the current one) in order to establish a constraint satisfying solution. This result is especially relevant for global reconfiguration, as local reconfiguration is already very efficient.

The conclusion is that it is not needed to perform the QoS-optimisation phase after a parameter event for this WSN model, as the Pareto set does not change. We may keep the current parameter vectors, and only need to recompute the metrics and check for constraint satisfaction. In general, however, we may not draw this conclusion. Determining the precise relationship between uncontrollables and Pareto optimality is an interesting topic for future work. It is, for example, interesting to know under which conditions the Pareto set is insensitive to an uncontrollable.

6.5.3 Broken Nodes

As it turns out that parameter events are not a major issue for the WSN task we study, we repeat the experiments of the previous sub-section for the next scenario: broken nodes. For each of the same 100 networks of 900 nodes, we establish an initial Pareto set $\mathcal{C}_0$. We then remove ten random nodes and repair the tree as proposed in Section 6.2. Then, we recompute the metrics of the configurations in $\mathcal{C}_0$ and call the new configuration set $\mathcal{C}_1$. This set represents the "do nothing" option. Subsequently, we use efficient global parameter reconfiguration to find the Pareto-optimal configuration set $\mathcal{C}_2$ in the new network with ten nodes fewer, and compute the quality difference between $\mathcal{C}_1$ and $\mathcal{C}_2$.

Also in this scenario it turns out that the quality differences are really small: about 0.001 on average. This suggests that the impact of nodes leaving the network and locally patching the routing tree on a properly configured WSN is quite small. Apparently, the changes that occur in the tree are not very significant on the scale of the network as a whole that mostly stays intact. Therefore, parameter reconfiguration after repairing the routing tree does not seem to be needed. Naturally, these conclusions can only be drawn for the WSN task and model that we study. The behaviour of other tasks and models needs to be verified on a case-by-case basis.

6.5.4 Mobile Sink

In this section, we examine the method of adaptation while the sink is moving, as introduced in Section 6.3. Compared to the broken-nodes scenario of the previous sub-section, a mobile sink has a much larger impact on the routing tree, and we therefore expect a significant effect on the quality of the configurations if no parameter reconfiguration is performed. We therefore more thoroughly investigate this case and the various reconfiguration strategies.

The evaluation metrics used to compare the solutions are as follows:

- **Disruption time**: the duration of service disruption just after a topology event until the tree has been reconfigured to include all nodes. This is equal to the time needed for QuickFix to complete. During Controlled Flooding (if CF continues after QuickFix finishes) and parameter optimisation, the network does function, though its service quality is reduced. Hence, for the parameter and criteria events discussed before, the disruption time is always zero.

- **Tree-reconfiguration time**: the total time needed to reconstruct the tree, comprising QuickFix,

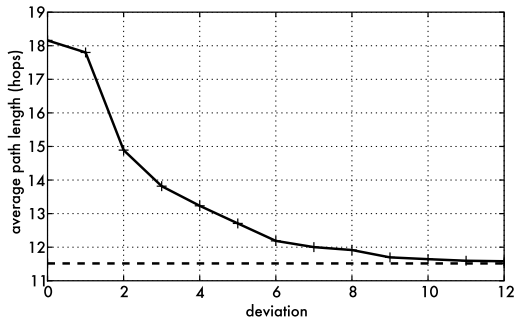
Controlled Flooding, and node-degree reduction.

- **Reconfiguration time**: the total duration of the tree- and parameter-reconfiguration process. The total reconfiguration time is also a rough indication of the amount of processing needed on the nodes (the optimisation time is dominated by processing).
- **Communication cost**: the average number of bytes transmitted for reconfiguration, over all nodes in the network.
- **Value loss**: the relative loss in value compared to the best case.

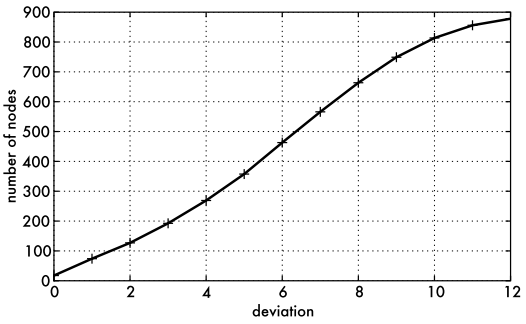
The simulated networks and WSN task are the same as before (900 nodes in an area of 300 × 300 m). The SinkMove timer was set to expire after 0.01 s, while the remaining set-up was as in Chapter 5. First, we look at the sink placed at coordinates (100,150), and the network configured with the method of the previous chapters. Then the sink moves to position (200,150), and the network is reconfigured using the various options described in this chapter. We simulated 100 different networks and report the medians of the metrics of interest. In the second scenario, the sink makes multiple consecutive moves, while the network is reconfigured after each move.

Tree Reconstruction. We first study the behaviour of the tree-reconstruction algorithm for a mobile sink described in Section 6.3 in the single-move scenario. All 100 networks were tested with various values of the deviation parameter, as well as full flooding (global reconfiguration) as a baseline. The degree-reduction algorithm with a target node degree of 2 was executed on the resulting networks. The first point to note is that the tree was correctly rebuilt in all of the cases. Figure 6.9(a) shows that average path length decreases monotonically from almost 18.2 to 11.5 when increasing the deviation from 0 (only QuickFix) to 12. The optimal average path length (when fully flooding the network) is also 11.5. Figure 6.9(b) indicates that the size of the affected region also grows steadily with the deviation until, at deviation 12, almost the whole network is reconfigured, and hence we obtain an approximate SPST with this deviation (within the degree constraint). Observe that the active area first grows faster than linearly, but slows down after deviation 6; this is the point where Controlled Flooding reaches the edges of the network.

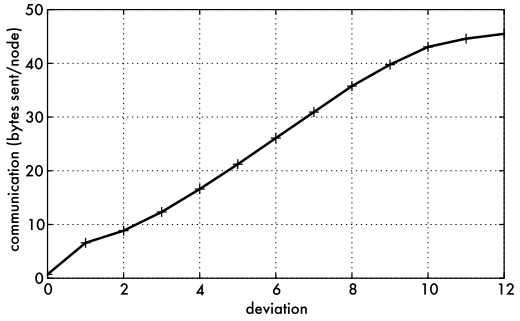
Along with the active area, the amount of communication increases in a similar pace (Figure 6.9(c)). As expected, also the total tree-reconfiguration time, excluding degree reduction, increases with the deviation (Figure 6.9(d), dashed line), with an offset due to QuickFix. The time needed for QuickFix/CF is relatively short compared to the time used to reduce the node



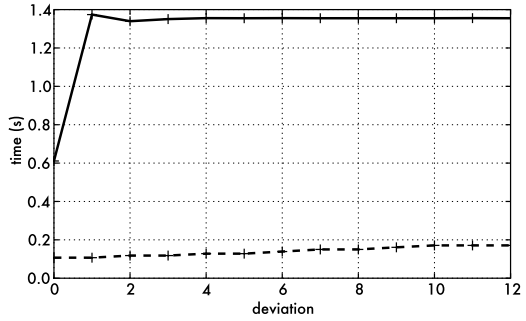
(a) Average path length (the dashed line is the optimum)



(b) Size of the active area



(c) Communication cost

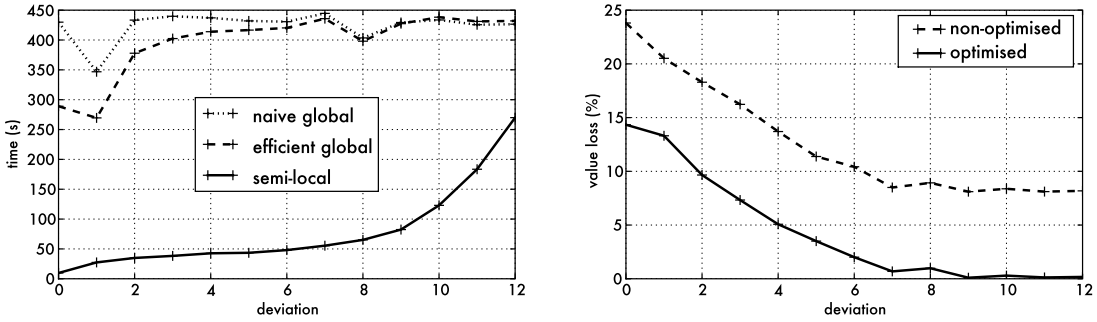


(d) Total tree-reconfiguration time (solid line) and only QuickFix+CF (dashed line)

Figure 6.9: Evaluation of tree reconstruction for various deviation values.

degrees, and together with the fact that the latter does not depend on the size of the active area, this explains why the total time spent on tree reconstruction (solid line in Figure 6.9(d)) is not clearly dependent on the deviation. Also recall that if only QuickFix is used, the degree-reduction algorithm is hardly effective due to the small active area, which is why the run time at deviation 0 is lower than for the others. Overall, the tree-reconstruction time is always less than 1.4 s, while the disruption time, comprising only the time need for QuickFix, is just over 0.1 s.

Parameter Optimisation. Now compare the optimisation times of the various cases of parameter optimisation in Figure 6.10(a). We confirm that efficient global reconfiguration, in which only nodes in the active area recompute their Pareto points, is always faster than the naive version, and this is most pronounced for small deviations. However, the differences are not as large as might be expected. Local optimisation (semi local to be precise, as explained in Section 6.4.2) on the other hand, in which the same nodes recompute their Pareto points as in the efficient-global case, but with boundary nodes using just one configuration for their child nodes outside the active area (instead of their full set of Pareto points), is much faster. This may imply that the configuration



(a) Optimisation time

(b) Value loss

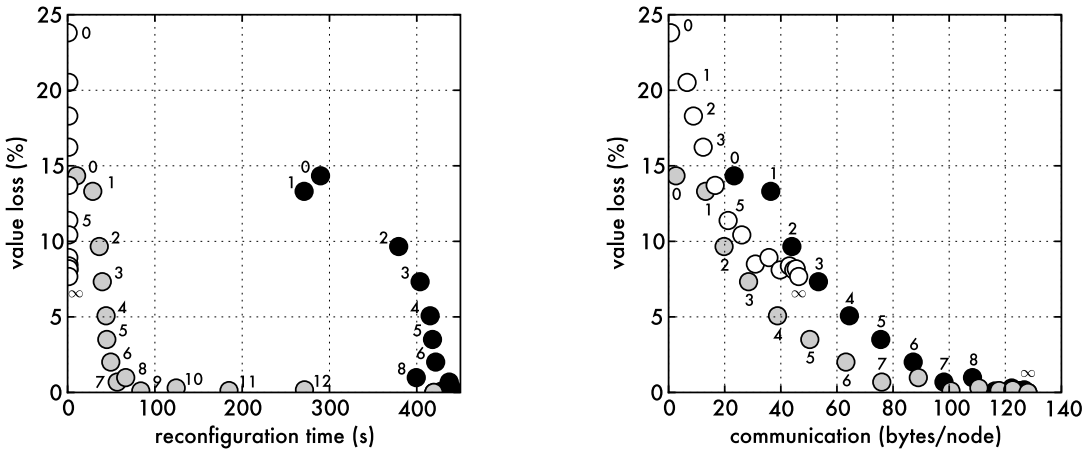
Figure 6.10: Timing results and value improvement for parameter optimisation.

sets of nodes closer to the sink are larger than those of nodes further away. It is interesting to see that the optimisation time of the localised algorithm initially increases very slowly (sub-linearly) with the deviation, starting at just 9.7 s. Deviation 5 appears to be an inflection point beyond which the rate of increase grows quickly. Eventually, the three lines meet at 418 s (about seven minutes; not visible in the graph), when fully flooding the network; this is equivalent to deviation infinity, as the active area is the whole network.

Next, we test the quality of the resulting configurations by comparing their values. The best value occurs when using full flooding and global parameter optimisation. Using this value as a baseline, Figure 6.10(b) shows the relative loss in value when using the other methods. It turns out that, for the target-tracking task, all methods for parameter optimisation, semi-local and global, achieve the same quality (the solid line), while not optimising is significantly worse (8 to 10 percentage points; the dashed line). Given its low overhead, this makes local reconfiguration very attractive for any deviation. The best value possible when not optimising (at deviation 12) can be attained with optimisation already at deviation 3. For a larger deviation, we see a steady improvement in value, which is consistent with the assumption that quality of the task improves if the average path length is reduced for a given degree target Δ, and vice versa (see Section 5.1). At deviation 0, the difference with the optimum is quite large at 14.3%, but after deviation 6 it becomes smaller than 1%.

6.5.5 Quality/Cost Trade-offs.

Combining all results yields the totals for the reconfiguration process in the evaluation metrics. The disruption time only depends on tree reconstruction (QuickFix) and has been reported above. Figure 6.11 gives an overview of the trade-offs between the total time and communication costs of



(a) Time vs. value loss

(b) Communication vs. value loss

Figure 6.11: Pareto plots for the reconfiguration process. Black dots belong to (efficient) global parameter reconfiguration, grey to (semi) local, and white to no parameter optimisation. Deviation values are given at interesting trade-off points.

reconfiguration, and the value loss of the resulting configuration, for all reconfiguration options. The two plots should be seen together as a three-dimensional trade-off space. It is immediately clear that all global-reconfiguration points (the black dots) are dominated by the semi-locally optimised (grey) and non-optimised (white) options. In contrast, most of the other points are Pareto optimal. As non-optimisation is obviously the fastest, it is the best choice when speed and low processing costs are most important, although the loss in value is at least 7.7%. In many cases, however, semi-local reconfiguration provides the best trade-off between the three metrics: low cost and good quality. At deviation 5, for example, semi-local reconfiguration takes 44.9 s (of which the service is disrupted for about 0.1 s), costs 50.3 bytes of communication per node, and the overall quality is 3.5% lower than the best case. The configuration space that was analysed in this time, for the active area of 350 nodes, has a size of 27^{350} configurations, of which the found configuration has the optimal value.

Multiple Moves of the Sink. It is interesting to see what happens to the loss of value when the sink moves repeatedly, and the network is locally optimised at each move. Figure 6.12 shows the value loss of semi-local optimisation with deviation 5 compared to the optimal case, for 25 consecutive moves of 50 m in random directions. The results are averages over five different runs. A very irregular pattern is visible, but the trend is a slowly increasing loss for each move. We therefore suggest to do a full network reconfiguration periodically, or when the attained value becomes too

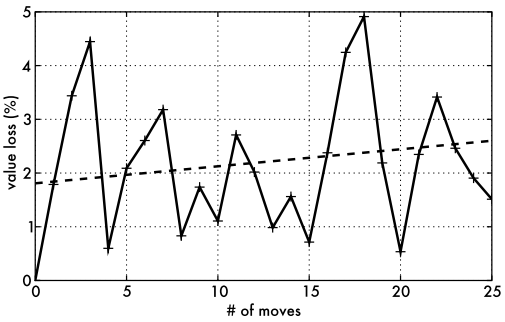


Figure 6.12: Multiple consecutive moves: value loss compared to optimal with trend line.

low.

Discussion. The best choice of reconfiguration method and deviation value in the mobile sink case heavily depends on the WSN task and its requirements as well, and specifically on the sink's behaviour. Due to the unpredictable nature of the sizes of the Pareto sets and therefore the optimisation time, as well as the quality of the resulting task-level Pareto set, it is hard to give guidelines for this choice. In practise, a system could first have a calibration phase to tune the deviation value, or simulations like ours can be used.

The methods that do all processing in-network are useful for applications in which the sink stays at its position for a while before moving again (e.g. when moving the sink for lifetime improvement), to justify the cost and speed of reconfiguration. For scenarios such as disaster recovery, in which the sink (rescue worker with handheld) may move a bit faster, an interesting option is to deploy special, more powerful nodes (such as handhelds) that handle most of the optimisation duties, or even do all the work at the sink. This is again a trade-off: between communication and processing cost (offloading computation effort increases the amount of communication between sensors and sink), and of course the cost of the additional nodes. For example, doing the QoS analysis for deviation 5 as above on a laptop (Intel Core 2 Duo processor at 2.4 GHz) takes 3.0 s for the globally-optimal case, and just 0.6 s for the localised case. For handheld devices, these numbers would be higher, but still much lower than when done in-network on sensor nodes. However, the communication costs per node increase by about five times (both global and local). Future work will have to focus on this trade-off in more depth.

6.6 Case Study: Building Monitoring

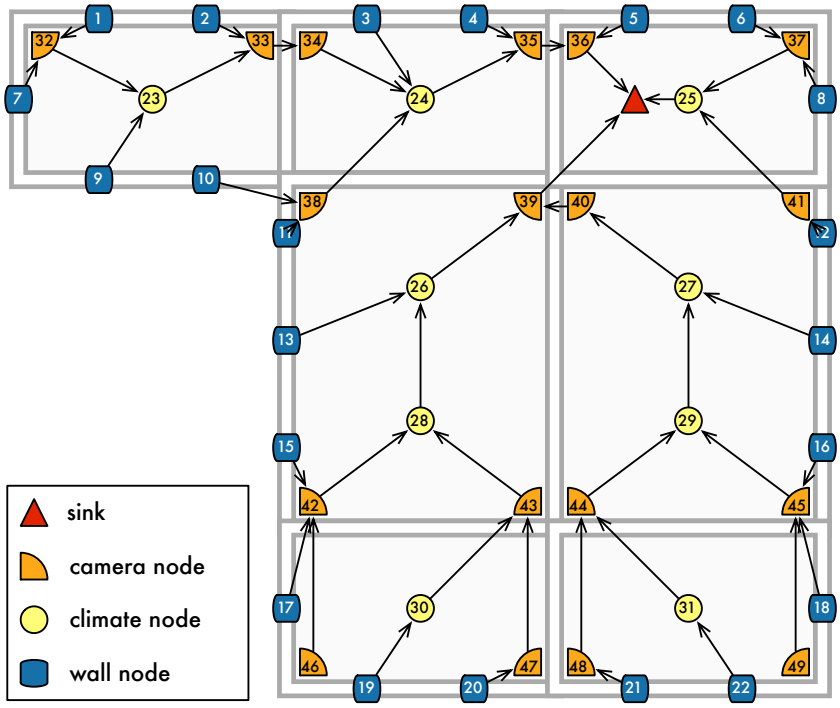
This section shows another example to illustrate the use of the configuration method in a different scenario. The main aim of this section is to highlight its support for heterogeneous collections of nodes, having various capabilities and parameter sets. We look at a wireless sensor network for monitoring a large building.

6.6.1 Situation and Model

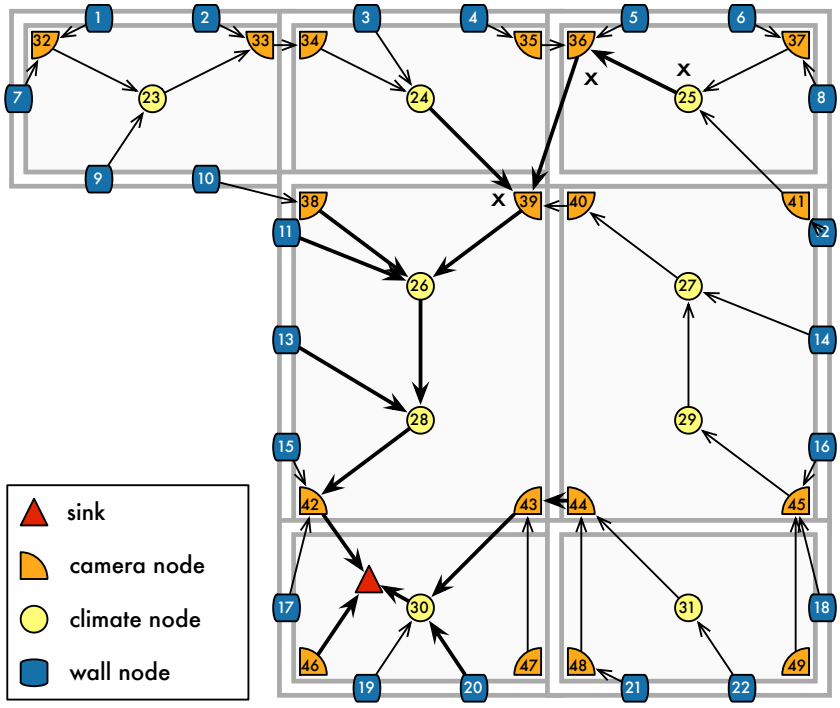
Figure 6.13(a) shows a map of one of the floors of a hospital building that is monitored by a sensor network. Our goal is to configure this network. As our hospital is located in an earthquake-prone area, structural monitoring is used to detect weak spots in the walls of the building, such that early measures can be taken to avoid damage and its possible catastrophic effects. Sensors that measure, for example, vibrations [34] or moisture are placed at or inside the walls of the building. Secondly, we have cameras inside the rooms of the hospital to keep an eye on the patients, which operate on a low frame rate. Besides that, we also monitor the rooms' climate – temperature and humidity – by one or two sensor nodes in each room. We assume that all measured information is collected by a mobile handheld device: the sink. Figure 6.13(a) shows the placement of the various types of nodes in the area.

The operator responsible for monitoring the building holds on to the sink node and is able to select one camera node at a time to watch its video stream. Wall and climate nodes periodically take measurements, which are forwarded to the sink. We assume no reliable data transport (no retransmissions) is used. The wall nodes are less powerful nodes and are assumed to be located in less accessible places (that is, changing batteries is hard). Therefore, these nodes are considered special and are not used to relay any data for other nodes. The network set-up is similar to IEEE 802.15.4 networks with their distinction between full-function devices and reduced-function devices, arranged in a cluster-tree network. The cluster-tree is in fact just a tree in which the reduced-function devices area always leaf nodes. This also means that our configuration method applies to it; tree construction can easily enforce reduced-function devices to be tree leaves by making them not forward Flood messages.

We explore a different method for medium access in this example, which is based on TDMA (time-division multiple access), and show that our configuration method is particularly suited to this kind of networks. TDMA has the property that transmissions can be cleanly separated, thereby avoiding collisions. This implies that parts of the network can be considered independently from



(a) Initial situation with routing tree.



(b) After a move of the sink. Tree repaired with $dev = 0$. Nodes marked by an **x** were lost and reconnected. Changed links are drawn with heavier lines.

Figure 6.13: Building-monitoring case study.

one another, which is an ideal situation for our configuration method, which assumes clusters to be independent entities (apart from their composition in the cluster hierarchy). The disadvantage of TDMA is the overhead caused by the need for synchronisation between nodes.

One of the challenges in using TDMA for sensor networks is the assignment of time slots to nodes for their transmissions, while keeping the total length of the schedule small. The problem is related to the classical problem of graph colouring, for which there are no known algorithms that construct a minimum-length schedule in an efficient way (the problem is NP-complete) [54]. However, there are (distributed) algorithms available that efficiently provide good solutions [28, 37, 41], for example based on a greedy construction of the schedule, and are also able to repair the schedule at run time in case changes in the network topology occur.

We assume that one of the above algorithms and a mechanism for time synchronisation are in place and a so-called *broadcast schedule* has been established soon after deployment. The schedule is periodic, and the schedule for each node contains a time slot for its own broadcast, as well as time slots for each of the nodes that are close enough to interfere with the node's transmissions. This means that the schedule contains slots for each of the nodes in a two-hop neighbourhood. The schedule ensures that the node's broadcasts never collide with other nodes' broadcasts. When the network is in operation, the radio of a node needs to be in receive mode only during the time slots of its child nodes in the routing tree. An additional sleep period may be introduced in the TDMA schedule to allow nodes to sleep more.

We consider the following quality metrics for our configuration problem (all are greater-is-better metrics):

1. **Video quality** (q): the average rate of the video stream in number of images per minute over all nodes.

2. **Speed** (s): inverse of the average latency from sensor to sink in seconds.

3. **Wall-measurement rate** (wm): average number of wall-measurement samples taken per hour over all wall nodes.

4. **Climate-measurement rate** (cm): average number of climate-measurement samples taken per hour over all climate nodes.

5. **Completeness** (c): fraction of the number of all measurements/samples taken in the network that arrive at the sink.

6. **Wall-node lifetime** (wl): time in hours until the first wall node runs out of battery.

7. **Other nodes lifetime** (ol): average battery life in hours of climate and camera nodes.

These metrics are task-level metrics, as well as cluster-level metrics for any cluster in the network (according to Definition 3.1) including one-node clusters. Note that because of the special nature of wall nodes, we have split the lifetime metric into two separate metrics. The first one is a minimum-lifetime metric that considers each wall node to be essential. This metric pushes the QoS optimiser to balance the workload evenly over all wall nodes. For the remaining nodes, an average lifetime metric is defined. Separating the lifetime metrics makes it easier to set priorities and constraints, and change them if deemed necessary.

For the speed metric, we assume that the per-hop delay is equal to the length of the TDMA period, and for simplicity in this example, that this period is the same for all nodes. We also assume that there is no queueing delay at the nodes, which is justified by the fact that the rates are low. Hence, the speed is inversely proportional to the hop count.

The parameters of the nodes and their values are defined in Tables 6.3–6.5. The parameter set of a wall node (all combinations of parameter values) has 9 parameter vectors, while climate and camera nodes both have 12. We assume that the sink node is not configurable. Note that the parameters and values for any node may be completely different, thereby allowing very heterogeneous networks. The full configuration space has size $9^{22} \cdot 12^{27} = 1.37 \cdot 10^{29}$.

Mappings from parameters to cluster metrics, as well as cluster-to-cluster metrics are given in Tables A.1–A.4 in Appendix A. In this example, we skip the layer of node metrics, and go directly to cluster metrics, even for single nodes. We leave it to the reader to verify that these mappings are correct.

6.6.2 Configuration

After construction of the broadcast TDMA schedule, the configuration method commences as usual by constructing a routing tree, this time taking into account that wall nodes are always leaf nodes. We use a degree target $\Delta = 3$. The resulting tree is drawn in Figure 6.13(a).

The next step is QoS analysis. Running the algorithm of Chapter 4 gives the set of 15 Pareto points shown in Table 6.6. We observe a trade-off between most metrics, except for speed and completeness. Speed is constant, as it only depends on the hop count, and only the tree-construction phase affects it. Apparently, lower completeness values than 94% provide no good trade-offs.

Table 6.3: Wall-node parameters

Parameter	Set of values (quantity)
Sample rate (r in samples/h)	$\{5, 10, 15\}$
Transmission power (p in dBm)	$\{-25, -15, -10\}$

Table 6.4: Climate-node parameters

Parameter	Set of values (quantity)
Sample rate (r in samples/h)	$\{20, 40, 60\}$
Transmission power (p in dBm)	$\{-15, -10, -5, 0\}$

Table 6.5: Camera-node parameters

Parameter	Set of values (quantity)
Video quality (r in frames/min)	$\{10, 20, 30\}$
Transmission power (p in dBm)	$\{-15, -10, -5, 0\}$

Table 6.6: Pareto points for the situation as in Figure 6.13(a)

q	s	wm	cm	c	wl	ol
24	0.008	5.0	48.9	0.94	82255	41826
24	0.008	10.0	48.9	0.94	41127	41826
24	0.008	20.0	48.9	0.94	20563	41826
27	0.008	5.0	44.4	0.94	82255	41921
26	0.008	5.0	46.7	0.94	82255	42079
28	0.008	5.0	37.8	0.94	82255	46680
30	0.008	5.0	60.0	0.94	82255	28318
27	0.008	10.0	44.4	0.94	41127	41921
26	0.008	10.0	46.7	0.94	41127	42079
28	0.008	10.0	37.8	0.94	41127	46680
30	0.008	10.0	60.0	0.94	41127	28318
27	0.008	20.0	44.4	0.94	20563	41921
26	0.008	20.0	46.7	0.94	20563	42079
28	0.008	20.0	37.8	0.94	20563	46680
30	0.008	20.0	60.0	0.94	20563	28318

Table 6.7: Pareto points for the situation as in Figure 6.13(b)

q	s	wm	cm	c	wl	ol
30	0.008	5.0	60.0	0.92	82255	27870
30	0.008	10.0	60.0	0.92	41127	27870
30	0.008	20.0	60.0	0.92	20563	27870

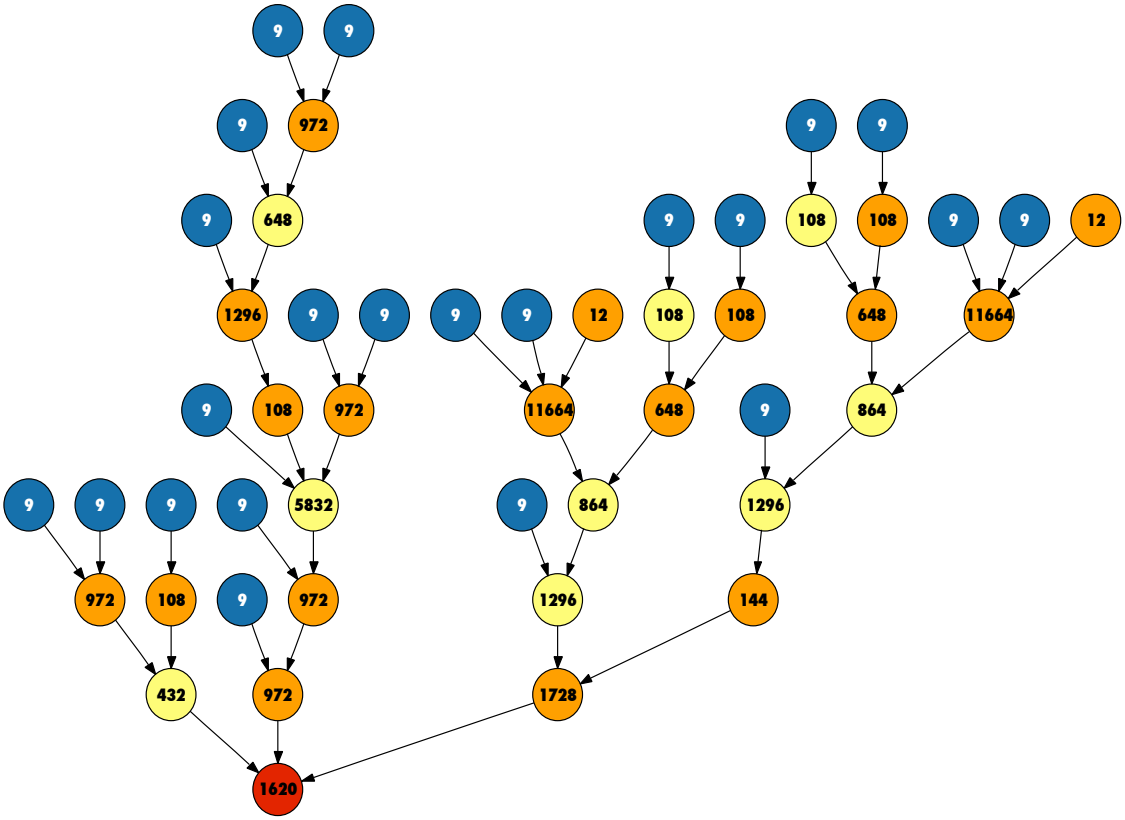


Figure 6.14: Processing costs per node for configuring the example of Figure 6.13(a). The numbers correspond to the size of the product set, or the size of the parameter set for leaf nodes, which is assumed to be proportional to the processing time. The colours of the nodes correspond with the colours in Figure 6.13

Figure 6.14 gives an overview of the complexity of running the QoS optimiser on this example. The figure shows the routing tree with at each node the size of the product set, or the size of the parameter set for leaf nodes, which is assumed to be proportional to the processing time. For example, for TelosB sensor nodes, the processing time in seconds is about $2.34 \cdot 10^{-2}$ times this number (see Section 4.6.4). For a laptop with a Core 2 Duo processor at 2.4 GHz, the factor was measured to be about $1.8 \cdot 10^{-5}$. The sum of the numbers in the graph, 46,374, is a measure of the QoS-optimiser's run time when centrally executed. This would take less than a second on the said laptop. When executed in a fully-distributed way, the longest path in the graph is what counts. This is the right-most path, yielding a total of 17,328, which would take about 405 s to execute on a network of only TelosB nodes. As we know from Chapter 4, distributed execution has the benefit of having much lower communication costs, as each node has to send only a single value to each of its child node in the loading phase, instead of parameters for all of its descendants. Further, if we equip our camera, climate and sink nodes by slightly more advanced processors, it is easy to get the configuration time down to less than a minute.

However, it is also possible to combine centralised and distributed computation. We stated above that wall nodes are less powerful and should be off-loaded as much as possible. We can do this by making their parents compute the configurations for them, and thereby following the concept of the cluster-tree network with full- and reduced-function devices. This is a straightforward option in the configuration method. The computation would now take a little bit longer (17,346) due to the reduced parallelism.

After finding the Pareto points, one of them is selected based on constraints and a value function, and loaded into the network. After that, normal operation of the network may start.

6.6.3 Moving Sink

If the sink moves to the other side of the building, the tree will be broken. Running the tree-reconstruction algorithm of this chapter with deviation value 0 (only QuickFix), we are able to repair the routing tree, as shown in Figure 6.13(b). Affected links are drawn with heavier lines in this picture. The new Pareto points are given in Table 6.7.

6.7 Summary

Wireless Sensor Networks are typically liable to changes at run time due to events in their environment. The configuration process described before this chapter did not deal with these

events, but rather configured the network for a static situation. This chapter introduces adaptation methods that intend to update the configuration in response to events at run time.

Three kinds of events are introduced, which require various degrees of change to the configuration. The least impact have changes in the objectives and/or constraints specified by the user, the so-called criteria events. Under such events, the current Pareto sets are still optimal, and reconfiguration simply means selecting a point using the new criteria, and loading it into the network.

Secondly, a parameter event involves a change in one or more uncontrollable parameters. After such an event, it is likely that the current Pareto sets are not valid anymore and need to be recomputed. Luckily, it is normally not necessary to re-analyse the whole network in order to find up-to-date globally Pareto-optimal configurations. Because of the hierarchical nature of the configuration process, only the Pareto points for clusters that contain changes need to be recomputed. Details are given on how this works.

The event that has the largest impact is the topology event. As part of such an event, nodes may move, appear or disappear, or communication links may become dysfunctional. The effect is that the routing tree may be broken and therefore needs to be fixed. We provide tree-reconfiguration mechanisms for the several kinds of topology events that may occur. These mechanisms are based on the tree algorithms of Chapter 5. A special case that we treat in more detail because of its practical relevance and huge impact on the configuration is the mobile sink.

If events occur regularly, global reconfiguration after each event may be too expensive. Instead, it is possible to reconfigure only a local region of nodes, around the place where the event occurred. The drawback is that the computed configurations are no longer guaranteed to be Pareto-optimal on a global level, and hence the achieved task quality may be lower than possible. This is again an example of a quality/cost trade-off.

Experimental evaluation shows that, for the example WSN task, parameter events do not have a big influence on the Pareto sets. This means that before and after the event, the same parameter vectors are Pareto optimal. However, the quality metrics may still change, and therefore also constraints may have been violated. The implication is that QoS optimisation is not really needed in this example, but the selection and loading phases still are (loading only if the new configuration is different from the current). An interesting direction for future work is to find out what exactly causes the Pareto points to be insensitive to changes in an uncontrollable.

A similar result is seen in the experiments for the topology events in which several nodes break

or run out of energy. What is always needed in these cases is a reconfiguration of the part of the tree where the event occurred. Simulations show, however, that the Pareto set does not change significantly for the example WSN task. That is, the current configuration is already a Pareto point, or nearly as good as one. Again, however, metrics would have changed after the event, and constraints need to be checked.

The mobile sink experiments show the real use of parameter reconfiguration. Since the tree changes significantly, also the Pareto points do. Simulations show that local reconfiguration of the tree as well as the parameters makes a lot of sense in this case, as the attained quality is comparable to the quality achieved by global reconfiguration, while the costs are much lower. Not reconfiguring leads to a much lower quality. By adjusting the size of the locality that is reconfigured, the quality/cost trade-off can be controlled.

This chapter completes our treatment of the WSN configuration problem.

Chapter 7

Conclusions

Numerous developments on Wireless Sensor Networks (WSNs) have been made since their emergence around ten years ago. A large variety of hardware for an even wider range of applications are readily available at the moment, efficient networking and data dissemination algorithms have been devised, while optimising specific properties of interest. Since WSNs typically contain a very large number of nodes, and each of these nodes has its own hardware or software settings, it is a huge challenge to configure each node such that the network behaves and performs according to the wishes of its owner. This is especially true if the demands are multifarious and inherently conflicting.

This thesis provides a methodology to configure WSNs such that constraints on multiple quality metrics (performance characteristics) are met, and the overall quality (performance) is optimised. The method is intended for networks with a single data sink, using a routing tree for communication. The overall quality is defined by a value function over all quality metrics. The configuration process is efficient, and scalable to very large networks. Furthermore, we provide ways to adapt the configuration at run time to changes in the environment of the network, or in the demands from the user.

7.1 Overview of the Configuration Method

The network may comprise a heterogeneous set of nodes (devices having various types and capabilities) and a task that is defined in terms of a number of high-level quality metrics. The method is specifically intended for networks featuring a single data sink, in which a tree topology is used for communication between the sensor nodes and the sink, where the sink is the root of the

tree. Furthermore, for the configuration process to be scalable, it needs to be possible to divide the network into a hierarchy of clusters (groups of nodes forming a sub-tree of the routing tree), such that each cluster has its own quality metrics. The hierarchy implies that a larger cluster includes several smaller clusters, and that the larger cluster's metrics can be derived from these smaller cluster's metrics (for the precise requirements, see Chapter 4). We specified two example tasks – spatial mapping and target tracking – that fulfil these rules and can be configured efficiently for any network size.

The configuration process first collects information form the network if needed, and subsequently it builds the routing tree. We designed new algorithms to build a tree in which paths are as short as possible within a maximum node-degree constraint, the degree target Δ (see below for an explanation of the importance of Δ). The next, and most significant phase in the configuration process is the QoS-optimisation phase, in which the Pareto-optimal configurations for the given quality metrics are determined. This phase relies on the above cluster hierarchy. Subsequently, the Pareto points that meet the quality constraints are pulled through the value function, after which the best configuration is selected for use. This configuration is then loaded into the network.

All phases of the configuration process can be implemented and executed in either a centralised or a distributed way. The choice between centralised and distributed does not affect the quality metrics, but is important for the different aspects of configuration cost: time, processing costs per sensor node, and communication costs per sensor node. The distributed methods generally have a better time complexity and lower communication costs. However, the actual run times can be significantly longer (especially for the distributed QoS optimiser) due to the limited processing capabilities of the sensor nodes, and the centralised algorithms do not require processing on the energy-constrained sensor nodes.

The effort required to adapt the configuration to changes in the environment or objectives can usually be restricted to a local region around the occurrence location of the change, without giving up too much in quality. Reconfiguration may be triggered by a change in the constraints or value function due to renewed priorities of the user. In this case, the current set of Pareto-optimal configurations is still valid, and may be reused for selecting a new configuration. A so-called parameter event occurs when a property of the environment changes, for example the quality of a link, without breaking the routing tree. Pareto points may now change, and should therefore be recomputed. Finally, a topology event may break the tree, and needs all configuration steps to be performed again. A special case of a topology event that is studied in detail, and for which we

Table 7.1: Handles to control the quality/cost trade-off

Product-set threshold	The maximum size of the product set per iteration of the QoS-optimisation algorithm may be limited to any threshold. The result is that the worst-case time complexity of the optimiser is linear, though it is no longer guaranteed that the resulting configurations are Pareto optimal. A smaller threshold generally implies a lower configuration cost and a lower task quality, and vice versa.
Degree target Δ	A lower maximum node degree in the network leads to a significantly lower time complexity of the configuration process. Additionally, it has a positive effect on certain task quality metrics, due to improved load balancing. However, forcing the tree to have lower degrees tends to make the average path length large, which deteriorates other quality metrics.
Locality, deviation dev	Locality only plays a role when adapting an already configured WSN to a new situation. The deviation parameter dev controls the size of the area that is reconfigured. The larger the size of this area, the better the resulting task quality and the larger the cost of reconfiguration.

developed a dedicated tree-reconstruction scheme, is the mobile sink case.

Note that our design objectives for the configuration process are twofold and inherently conflicting: the task's quality, as well as the cost of configuration should both be optimised. Our solutions are aware of this quality/cost trade-off within the configuration process, and provide handles to choose a suitable point in the trade-off space. Table 7.1 gives an overview of these handles and how they influence the trade-off. The best trade-off depends on many factors, including the nature of the task, the environment of the network, and the wishes of the user. In this thesis, we therefore merely present the handles and their effect, and rely on the user to select the proper settings.

7.2 Recommendations for Future Work

While this thesis provides a complete and efficient solution for the configuration problem for the given class of WSNs, there is room for extension. Below are some ideas for future work.

- Our QoS optimiser has been designed for networks with a routing tree in place. However, the correctness of the incremental optimisation method has been defined in more general terms, and may therefore also apply to other routing techniques [4]. Future research could therefore focus on supporting alternative routing protocols.

- The current method first optimises the routing tree, and then the remaining parameters.

Certain points in the configuration space that have the *parent node* as a parameter may therefore be missed. Ideally, the configuration process would jointly optimise the tree (*parent node*s) and the other parameters. Due to the required leaf-to-root cluster order of the QoS optimiser, which needs a tree to start, this seems to be impossible. However, it may still be interesting to revisit this issue and study possible alternatives.

- Experiments in Chapter 4 show that the QoS optimiser is scalable for the example networks, which are considered to be representative and accurate instances of typical WSN tasks. However, in general, the worst-case optimisation time is still exponential in the size of the network. Scalability essentially relies on a relatively small number of Pareto points in each of the clusters that is used in the algorithm. It may be possible to specify a class of WSN task models for which the algorithm is guaranteed to be scalable. In such a class, the mapping functions and values of the parameters would be restricted.

- The option to exploit the benefits of both the centralised and distributed implementations of the configuration process by deploying powerful, dedicated configuration nodes, or moving the computation form sensor nodes to already existing high-capacity nodes, seems very promising. Such a scheme fits almost readily in the configuration method as it is, and deserves experimental evaluation.

- While the resource metrics and constraints are already integrated in the QoS optimiser, we did not yet take these into account in the experiments. Resource metrics are especially important when looking to run multiple tasks on one WSN simultaneously. It is quite straightforward to include a resource model for a resource that is local to a node, such as the available clock cycles for processing on the micro-controller. However, designing a resource model for a resource that is shared between nodes, such as the wireless communication channel, and its integration in the QoS optimiser is challenging, as the monotonicity of the hierarchical method should be ensured. Hence, the design and integration of such resource models would be very interesting.

- Section 4.5 briefly touched on the topic of configuring a WSN to run multiple tasks concurrently. The current method is able to support this, if the parameters and metrics of all tasks are fused in a single configuration space, and all tasks share the same routing tree. A more general approach for sharing the WSN as a platform between independently running tasks is formulated as a multi-dimensional multiple-choice knapsack problem (MMKP).

Working out the details of a solution to this problem is yet to be done. Good resource models, as hinted at in the previous point, are key to this approach.

- In Chapter 6 on adaptation, it was found that the Pareto-optimal configuration set is insensitive to changes in certain uncontrollable parameters. This is potentially a very powerful feature, as only the current Pareto-optimal configurations need to be considered after a shift in such an uncontrollable parameter, and reconfiguration can be done very efficiently. The precise relation between uncontrollables and their values, and the dominance relation of configurations, is still unclear. Identifying such uncontrollables and the ranges of values for which the Pareto set is invariant, would be a very useful next step.

- Throughout the thesis we have used the assumption from Section 3.3 that task quality and configuration cost are independent optimisation targets. In practise, however, this is not necessarily the case, and the configuration process may affect the quality of the running task, especially when reconfiguring relatively often. Future work could focus on integrating the optimisation of task quality and configuration cost for specific cases.

- Another possible extension is the use of probability distributions instead of deterministic mapping functions, probably obtained from experiments, in order to better assess the effects of inaccuracies in the mappings. This may require a probabilistic version of Pareto algebra.

- Finally, as the current experimental evaluation is mostly based on simulation (though the run time of a TinyOS implementation of the QoS optimiser was measured on real TelosB sensor nodes), a feasibility check of the configuration method on a real WSN is desirable.

Appendix A

Mappings for the Case Study

The appendix contains the mapping functions for the case study of Section 6.6. The mapping functions use the following helper functions from transmission power in dBm to respectively reliability and current draw in mA:

$$p2r(p) = \begin{cases} 0.60 & \text{if } p = -25 \\ 0.80 & \text{if } p = -15 \\ 0.90 & \text{if } p = -10 \\ 0.95 & \text{if } p = -5 \\ 0.99 & \text{if } p = 0 \end{cases} \tag{A.1}$$

$$p2i(p) = \begin{cases} 8.50 & \text{if } p = -25 \\ 9.90 & \text{if } p = -15 \\ 11.0 & \text{if } p = -10 \\ 14.0 & \text{if } p = -5 \\ 17.4 & \text{if } p = 0 \end{cases} \tag{A.2}$$

Table A.1: One-node-cluster mappings for a wall node n

Video quality	$q(n) = 0$	(A.3a)
Speed	$s(n) = 1$	(A.3b)
Wall-meas. rate	$wm(n) = r(n)$	(A.3c)
Climate-meas. rate	$cm(n) = 0$	(A.3d)
Completeness	$c(n) = p2r(p(n))$	(A.3e)
Wall-node lifetime	$wl(n) = \frac{E}{(T_\mathrm{s} I_\mathrm{s} + T_\mathrm{tx} \cdot p2i(p(n))) \cdot r(n)}$	(A.3f)
Other nodes lifetime	$ol(n) = 0$	(A.3g)

With E the battery power in mAh, T_s and I_s the sample time (in h) and current (in mA), and T_tx the transmission time.

Table A.2: One-node-cluster mappings for a climate node n

Video quality	$q(n) = 0$	(A.4a)
Speed	$s(n) = 1$	(A.4b)
Wall-measurement rate	$wm(n) = 0$	(A.4c)
Climate-measurement rate	$cm(n) = r(n)$	(A.4d)
Completeness	$c(n) = p2r(p(n))$	(A.4e)
Wall-node lifetime	$wl(n) = \infty$	(A.4f)
Other nodes lifetime	$ol(n) = \frac{E}{P}$	(A.4g)

With E the battery power in mAh, and

$$P = T_\mathrm{s} \cdot I_\mathrm{s} \cdot r(n) + T_\mathrm{tx} \cdot p2i(p(n)) \cdot (r(n) + f(n)) + \frac{nc(n) \cdot t}{T} \cdot I_\mathrm{rx},$$

with T_s and I_s the sample time (h) and current (mA), T_tx the transmission time (h), $f(n)$ an estimate of the rate at which messages from n's descendants are forwarded, $nc(n)$ the number of children of node n, I_rx the current drawn in receive mode (mA), and t and T the duration of a time slot and the period of the TDMA schedule (h).

Table A.3: One-node-cluster mappings for a camera node n

Video quality	$q(n) = r(n)$	(A.5a)
Speed	$s(n) = 1$	(A.5b)
Wall-measurement rate	$wm(n) = 0$	(A.5c)
Climate-measurement rate	$cm(n) = 0$	(A.5d)
Completeness	$c(n) = p2r(p(n))$	(A.5e)
Wall-node lifetime	$wl(n) = \infty$	(A.5f)
Other nodes lifetime	$ol(n) = \frac{E}{P}$	(A.5g)

With E the battery power in mAh, and

$$P = T_{\mathrm{s}} \cdot I_{\mathrm{s}} \cdot \frac{r(n)}{18} + T_{\mathrm{tx}} \cdot p2i(p(n)) \cdot \left(\frac{r(n)}{18} + f(n) \right) + \frac{nc(n) \cdot t}{T} \cdot I_{\mathrm{rx}},$$

with T_{s} and I_{s} the sample time (h) and current (mA), T_{tx} the transmission time (h), $f(n)$ an estimate of the rate at which messages from n's descendants are forwarded, $nc(n)$ the number of children of node n, I_{rx} the current drawn in receive mode (mA), and t and T the duration of a time slot and the period of the TDMA schedule (h). The rate $r(n)$ is divided by 18, the number of camera nodes, as only one video stream is requested at a time.

Table A.4: Cluster-to-cluster mappings for a cluster c

Video quality	$q_\Sigma(c) = \sum_{i \in sub(c)} q_\Sigma(i)$	(A.6a)
Speed	$s_\Sigma(c) = \frac{1}{1 + \sum_{i \in ch(c)} (s_\Sigma(i)^{-1} + 1)}$	(A.6b)
Wall-meas. rate	$wm_\Sigma(c) = \sum_{i \in sub(c)} wm_\Sigma(i)$	(A.6c)
Climate-meas. rate	$cm_\Sigma(c) = \sum_{i \in sub(c)} cm_\Sigma(i)$	(A.6d)
Completeness	$c_\Sigma(c) = c_\Sigma(rt(c)) \left(1 + \sum_{i \in ch(c)} c_\Sigma(i)\right)$	(A.6e)
Wall-node lifetime	$wl(c) = \min_{i \in sub(c)} wl(i)$	(A.6f)
Other nodes lifetime	$ol_\Sigma(c) = \sum_{i \in sub(c)} ol_\Sigma(i)$	(A.6g)

For combined cluster c, the root cluster is denoted $rt(c)$, the set of child clusters $ch(c)$; $sub(c) = \{rt(c)\} \cup ch(c)$. All metrics with sub-script Σ are cumulative metrics that need to be divided by the number of nodes to obtain the desired average values. The resulting speed value needs to be divided by the TDMA-period length to obtain the real speed in s^{-1}.

Bibliography

[1] M. M. Akbar, E. G. Manning, G. C. Shoja, and S. Khan. Heuristic solutions for the multiple-choice multi- dimension knapsack problem. In *International Conference on Computational Science*, May 2001.

[2] K. Akkaya and M. Younis. An energy-aware QoS routing protocol for wireless sensor networks. In *ICDCSW 2003, Proc.*, pages 710–715. IEEE, 2003.

[3] K. Akkaya and M. Younis. Energy-aware routing to a mobile gateway in wireless sensor networks. In *GlobeCom'04, Proc.*, pages 16–21. IEEE, 2004.

[4] K. Akkaya and M. Younis. A survey on routing protocols for wireless sensor networks. *Elsevier Journal of Ad Hoc Networks*, 3(3):325–349, May 2005.

[5] G. Baliga and P. Kumar. Middleware for control over networks. In *Conference on Decision and Control (CDC 2005), Proc.* IEEE, December 2005.

[6] S. Bhattacharya, G. Xing, C. Lu, G.-C. Roman, B. Harris, and O. Chipara. Dynamic Wake-up and Topology Maintenance Protocols with Spatiotemporal Guarantees. In *IPSN'05*, Los Angeles, CA, Apr. 2005.

[7] P. Boonma and J. Suzuki. MONSOON: A coevolutionary multiobjective adaptation framework for dynamic wireless sensor networks. In *Hawaii International Conference on System Sciences, Proc. of the 41st Annual*, pages 497–497. IEEE, January 2008.

[8] E. Cayirci and T. Coplu. SENDROM: Sensor networks for disaster relief operations management. *Wireless Networks*, 13(3):409–423, June 2007.

[9] A. Cerpa and D. Estrin. ASCENT: Adaptive Self-Configuring sEnsor Networks Topologies. In *IEEE INFOCOM'02*, volume 3, pages 23–27, June 2002.

[10] D. Chen and P. K. Varshney. QoS support in wireless sensor networks: A survey. In *Int. Conference on Wireless Networks (ICWN 2004)*. CSREA Press, June 2004.

[11] C.-Y. Chiang, R. Chadha, G. Levin, S. Li, Y.-H. Cheng, and A. Poylisher. AMS: An adaptive middleware system for wireless ad hoc networks. In *Military Communications Conference, 2005. MILCOM 2005. IEEE*, pages 1–7, Oct 2005.

[12] O. Chipara, Z. He, G. Xing, Q. Chen, X. Wang, C. Lu, J. Stankovic, and T. Abdelzaher. Real-time power-aware routing in sensor networks. In *IWQoS '06, Proc.*, pages 83–92, June 2006.

[13] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, USA, 2nd edition, 2001.

[14] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The runes middleware for networked embedded systems and its application in a disaster management scenario. In *Percom '07, Proc.* IEEE, 2007.

[15] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis. A reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, June 2007.

[16] Crossbow Technology. Telosb datasheet, 2007.

[17] G. Cugola and M. Migliavacca. A context and content-based routing protocol for mobile sensor networks. In *European Conference on Wireless Sensor Networks (EWSN '09), Proc.*, pages 69–85. Springer, 2009.

[18] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Evolutionary Computation, IEEE Transactions on*, 6(2):182–197, 2002.

[19] F. Delicato, P. Pires, L. Rust, L. Pirmez, and J. de Rezende. Reflective middleware for wireless sensor networks. In *Symposium on Applied Computing (SAC), Proc.*, pages 1155–1159. ACM Press New York, NY, USA, 2005.

[20] M. Demmer and M. Herlihy. The arrow distributed directory protocol. In *Distributed Computing (DISC'98), Proc.*, pages 119–133. Springer, 1998.

[21] M. Fürer and B. Raghavachari. Approximating the minimum-degree steiner tree to within one of optimal. *J. Algorithms*, 17(3):409–423, 1994. ISSN 0196-6774. doi: http://dx.doi.org/10.1006/jagm.1994.1042.

[22] M. Geilen and T. Basten. A calculator for Pareto points. In *Design, Automation and Test in Europe (DATE), Proc.*, pages 285–291, Los Alamitos, CA, USA, April 2007. IEEE Computer Society Press. See www.es.ele.tue.nl/pareto.

[23] M. Geilen, T. Basten, B. Theelen, and R. Otten. An algebra of Pareto points. *Fundamenta Informaticae*, 78(1):35–74, 2007.

[24] S. Haykin. *A Introduction to Analog and Digital Communications*. John Wiley & Sons, 1989.

[25] T. He, C. Huang, B. Blum, J. Stankovic, and T. Abdelzaher. Range-free localization schemes for large scale sensor networks. In *MobiCom 2003, Proc.* ACM, 2003.

[26] T. He, J. Stankovic, C. Lu, and T. Abdelzaher. SPEED: A stateless protocol for real-time communication in sensor networks. In *ICDCS 2003, Proc.* IEEE, May 2003.

[27] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network Magazine Special Issue*, pages 6–14, Jan 2004.

[28] T. Herman and S. Tixeuil. A distributed tdma slot assignment algorithm for wireless sensor networks. In *Algorithmic Aspects of Wireless Sensor Networks*, volume 3121 of *Lecture Notes in Computer Science*, pages 45–58, 2004.

[29] C. S. Hiremath. *New Heuristic And Metaheuristic Approaches Applied To The Multiple-choice Multi-dimensional Knapsack Problem*. PhD thesis, Wright State University, 2008.

[30] D. Jourdan and O. de Weck. Multi-objective genetic algorithm for the automated planning of a wireless sensor network to monitor a critical facility. In *Proc. of SPIE – Sensors, and Command, Control, Communications, and Intelligence (C3I)*, volume 5403, pages 565–575, 2004.

[31] H. Karl and A. Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005.

[32] H. S. Kim, T. F. Abdelzaher, and W. H. Kwon. Minimum-energy asynchronous dissemination to mobile sinks in wireless sensor networks. In *SenSys '03, Proc.*, pages 193–204, New

York, NY, USA, 2003. ACM. ISBN 1-58113-707-9. doi: http://doi.acm.org/10.1145/958491.958515.

[33] R. Krishnan and B. Raghavachari. The directed minimum-degree spanning tree problem. In *21st Conference on Foundations of Software Technology and Theoretical Computer Science (FST TCS '01), Proc.*, pages 232–243, London, UK, 2001. Springer-Verlag. ISBN 3-540-43002-4.

[34] N. Kurata, M. Suzuki, S. Saruwatari, and H. Morikawa. Actual application of ubiquitous structural monitoring system using wireless sensor networks. In *World Conference on Earthquake Engineering*, Oct 2008.

[35] C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete QoS options. In *Real-Time Technology and Applications Symposium, Proc.* IEEE, June 1998.

[36] P. Levis. *TinyOS Programming (revision 1.3)*, Oct 2006. URL `http://www.tinyos.net`.

[37] E. L. Lloyd. Broadcast scheduling for tdma in wireless multihop networks. In I. Stojmenovi, editor, *Handbook of Wireless Networks and Mobile Computing*. John Wiley & Sons, 2002.

[38] J. Lu, F. Valois, D. Barthel, and M. Dohler. Fisco: A fully integrated scheme of self-configuration and self-organization for wsn. In *Wireless Communications and Networking Conference (WCNC) 2007*, pages 3370–3375. IEEE, 2007.

[39] J. Luo and J.-P. Hubaux. Joint mobility and routing for lifetime elongation in wireless sensor networks. In *Infocom 2005, Proc.*, 2005.

[40] J. Luo, J. Panchard, M. Piórkowski, M. Grossglauser, and J.-P. Hubaux. Mobiroute: Routing towards a mobile sink for improving lifetime in sensor networks. In *DCOSS 2006, Proc.*, 2006.

[41] J. Mao, Z. Wu, and X. Wu. A tdma scheduling scheme for many-to-one communications in wireless sensor networks. *Computer Communications*, 30:863—872, 2007.

[42] J. N. Morse. Reducing the size of the nondominated set: Pruning by clustering. *Comput. Oper. Res.*, 7:55–66, 1980.

[43] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. QoS-aware middleware for ubiquitous and heterogeneous environments. *IEEE Comunications Magazine*, 39(11):2–10, Nov 2001.

[44] T. Okabe, Y. Jin, and B. Sendhoff. A critical survey of performance indices for multi-objective optimisation. In *Evolutionary Computation, 2003. CEC'03*, 2003.

[45] G. Palermo, C. Silvano, and V. Zaccaria. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing*, 1(3), 2006.

[46] J. Panchard, S. Rao, T. Prabhakar, J. Hubaux, and H. Jamadagni. COMMONSense Net: A wireless sensor network for resource-poor agriculture in the semiarid areas of of developing countries. *Information Technologies and International Development*, 4(1):51–67, 2007.

[47] V. Pareto. *Manuale di Economia Politica*. Piccola Biblioteca Scientifica, Milan, 1906. Translated into English by Ann S. Schwier (1971), Manual of Political Economy, MacMillan, London.

[48] S. Pattem, S. Poduri, and B. Krishnamachari. Energy-quality tradeoffs for target tracking in wireless sensor networks. In *IPSN 2003, Proc.*, LNCS 2634, pages 32–46. Springer-Verlag, 2003.

[49] M. Perillo and W. B. Heinzelman. Providing application QoS through intelligent sensor management. In *Int. Workshop on Sensor Network Protocols and Applications (SNPA '03)*. IEEE, 2003.

[50] L. Pirmez, F. Delicato, P. Pires, A. Mostardinha, and N. de Rezende. Applying fuzzy logic for decision-making on wireless sensor networks. In *Fuzzy Systems Conference, 2007*, pages 1–6. IEEE, 2007.

[51] N. Pogkas, G. E. Karastergios, C. P. Antonopoulos, S. Koubias, and G. Papadopoulos. Architecture design and implementation of an ad-hoc network for disaster relief operations. *IEEE Transactions on Industrial Informatics*, 3(1):63–72, 2007.

[52] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Embedded networked sensor systems (SenSys '04), Proc.*, pages 95–107, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-879-2. doi: http://doi.acm.org/10.1145/1031495.1031508.

[53] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *MobiCom 2000, Proc.* ACM, 2000.

[54] S. Ramanathan and E. L. Lloyd. Scheduling algorithms for multi-hop radio. In *COMM'92*, pages 211–222. ACM, 1992.

[55] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, 2004.

[56] K. Römer, O. Kasten, and F. Mattern. Middleware challenges for wireless sensor networks. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):59–61, 2002. ISSN 1559-1662. doi: http://doi.acm.org/10.1145/643550.643556.

[57] M. A. Rosenman and J. S. Gero. Reducing the pareto optimal set in multicriteria optimization. *Engineering Optimization*, 8:189–206, 1985.

[58] C. Schurgers, V. Tsiatsis, and M. B. Srivastava. STEM: Topology Management for Energy Efficient Sensor Networks. In *IEEE Aerospace Conference 2002*, 2002.

[59] C. Shen, C. Badr, K. Kordari, S. Bhattacharyya, G. Blankenship, and N. Goldsman. A rapid prototyping methodology for application-specific sensor networks. In *Computer Architecture for Machine Perception and Sensing (CAMP '06), Proc.*, pages 130–135. IEEE, September 2006.

[60] H. Shojaei, A. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A parameterized compositional multi-dimensional multiple-choice knapsack heuristic for cmp run-time management. In *Design Automation Conference (DAC)*, pages 917–922. ACM, July 2009.

[61] K. Sohrabi, J. Gao, V. Ailawadhi, and G. Pottie. Protocols for self-organization of a wireless sensor network. *IEEE Personal Communications*, 7(5):16–27, Oct 2000.

[62] Stichting IJkdijk. IJkdijk website, 2009. URL `http://www.ijkdijk.nl`.

[63] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. A framework for evaluating design tradeoffs in packet processing architectures. In *39th Design Automation Conference (DAC 2002)*, pages 880–885, New Orleans LA, USA, June 2002. ACM Press.

[64] A. Varga. OMNeT++ simulator. www.omnetpp.org, 2008. URL `http://www.omnetpp.org`.

[65] W. Wang, V. Srinivasan, and K.-C. Chua. Using mobile relays to prolong the lifetime of wireless sensor networks. In *MobiCom 2005, Proc.*, pages 270–283. ACM, 2005.

[66] Y. Wang, D. Han, Q. Zhao, X. Guan, and D. Zheng. Clusters partition and sensors configuration for target tracking in wireless sensor networks. In *Embedded Software and Systems: First International Conference (Icess '04), Proc.* Springer, 2005.

[67] M. Wolenetz, R. Kumar, J. Shin, and U. Ramachandran. A simulation-based study of wireless sensor network middleware. *International Journal of Network Management*, 15(4):255–267, 2005.

[68] E. Yang, A. Erdogan, T. Arslan, and N. Barton. Multi-objective evolutionary optimizations of a space-based reconfigurable sensor network under hard constraints. In *ECSIS Symp. Bio-inspired, Learning, and Intelligent Systems for Security, Proc.*, pages 72–75. IEEE, 2007.

[69] H. Yang, F. Ye, and B. Sikdar. Swarm Intelligence based Surveillance Protocol in Sensor Network with Mobile Supervisors. In *IEEE VTC-Spring'05*, Stockholm, Sweden, May 2005.

[70] C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal. Fast Multi-Dimension Multi-Choice Knapsack Heuristic for MP-SoC Run-Time Management. *System-on-Chip, 2006. International Symposium on*, pages 1–4, 2006.

[71] Y. Yu, B. Krishnamachari, and V. Prasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18(1):15–21, Jan/Feb 2004.

[72] W. Zhang and G. Cao. Optimizing tree reconfiguration for mobile target tracking in sensor networks. In *IEEE INFOCOM'04*, 2004.

[73] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, Nov 1999.

[74] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Evolutionary Methods for Design, Optimisation, and Control*, pages 95–100. CIMNE, Barcelona, Spain, 2002.

[75] E. Zitzler, L. Thiele, M. Laumanns, C. M. Fonseca, and V. G. da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, April 2003.

List of Publications

This thesis was submitted to both the National University of Singapore and Eindhoven University of Technology in the joint PhD program of these universities. The same thesis, apart from minor amendments and a different layout, was published in October 2009 at Eindhoven University of Technology (ISBN 978-90-386-1981-1).

Other Publications

1. R. Hoes, T. Basten, C.-K. Tham, M. Geilen, and H. Corporaal. Quality-of-service trade-off analysis for wireless sensor networks. *Performance Evaluation*, volume 66, number 3–5, pages 191–208, Elsevier, March 2009.

2. R. Hoes, T. Basten, W.-L. Yeow, C.-K. Tham, M. Geilen, and H. Corporaal. QoS management for wireless sensor networks with a mobile sink. In *European Conference on Wireless Sensor Networks (EWSN '09), Proc.*, pages 53–68, Cork, Ireland, Feb 2009. Lecture Notes in Computer Science 5432. Springer, Berlin, Germany, 2009.

3. R. Hoes, T. Basten, C.-K. Tham, M. Geilen, and H. Corporaal. Analysing QoS trade-offs in wireless sensor networks. In *10th ACM Symposium on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWiM), Proc.*, pages 60–69, New York, NY, USA, Oct 2007. ACM Press.

4. M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J.D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen. *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, chapter Dataflow Analysis for Real-Time Embedded Multiprocessor System Design, pages 81–108. Springer, May 2005.

5. H. Shojaei, A.H. Ghamarian, T. Basten, M. Geilen, S. Stuijk, and R. Hoes. A Parameterized Compositional Multi-dimensional Multiple-choice Knapsack Heuristic for CMP

Run-time Management. In: *Design Automation Conference (DAC)*, pages 917–922, July 2009. ACM Press.