

Delft University of Technology
Master's Thesis in Computer Science

Evaluating model-based diagnosis for wireless sensor networks

Adriaan de Jong



Evaluating model-based diagnosis for wireless sensor networks

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Adriaan de Jong
born in Geneva, Switzerland



Embedded Software Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.es.ewi.tudelft.nl

Evaluating model-based diagnosis for wireless sensor networks

Author: Adriaan de Jong
Email: adriaan@ch.tudelft.nl

Abstract

Model-based diagnosis is a technique where a model of a system is combined with observations from that system, to generate diagnoses for failures of the system. This thesis looks at how model-based diagnosis can be applied to wireless sensor networks (WSNs). WSNs are ad-hoc wireless networks of small form-factor, embedded nodes with limited memory and processor power. Further, they are often battery powered, meaning that energy use must be kept to a minimum. A diagnoser design is proposed that uses the distributed nature of WSNs to find initial symptoms based on a local model, while leaving the more complex computations required to combine these symptoms to a more powerful central sink computer. A proof-of-concept design is then implemented. Results from this implementation show that using model-based diagnosis in sensor networks is certainly a viable solution. The model used in the proof of concept application created during the work on this thesis did have some problems in dense networks, showing that care must be taken when crafting the model to ensure a successful deployment.

Thesis Committee:

Chair: Prof. dr. K.G. Langendoen, Faculty EEMCS, TU Delft
Committee Member: Prof. dr. ir. A.J.C. van Gemund, Faculty EEMCS, TU Delft
Committee Member: Dr. P.G. Kluit, Faculty EEMCS, TU Delft

Preface

This Master of Science thesis is the description of the work I did at the Embedded Systems group during the past year. My interest in wireless sensor networks came after following a seminar on the topic. The many different techniques used in wireless sensor networks, touching topics from a number of different specialisations in Computer Science, combined with the practical side of actual deployment, appealed to my own wide interests in the field. During discussions on possible research topics, the diagnosis problem came up, which after much research, resulted in the work you now see before you.

During the project, a number of people provide a great deal of support. First of all, I would like to thank Elisabeth for her patience during the sometimes long hours, and her support during the whole process. Further, I would like to thank my parents for their patience throughout all the years that it took to reach this point. I would also like to thank Gert-Jan, Otto, and Niels, who provided a great deal of support on the more technical side, especially when dealing with the testbed. Alex provided a great deal of support on the model-based diagnosis side of this paper. Finally, I would like to express a great deal of gratitude to Koen, who helped guide me throughout the whole process, providing excellent ideas and advice on some of the more difficult topics.

Adriaan de Jong

Delft, The Netherlands
22nd May 2009

Contents

Preface	v
Contents	vii
1 Introduction	1
1.1 Problem statement	2
1.2 Organisation	2
2 Background	3
2.1 Wireless sensor networks	3
2.2 Diagnosis techniques	4
2.3 Diagnosis in WSNs	5
3 Model-based diagnosis	7
3.1 The Model	7
3.2 Finding a diagnosis	8
4 Design of a model-based diagnosis framework for WSNs	13
4.1 Observations, predictions and environments	14
4.2 Model definition	14
4.3 Node-level inference engine	15
4.4 Creating a diagnosis	17
4.5 Discussion	18
5 Implementation	19
5.1 TinyOS	19
5.2 Observation and environment identifiers	20
5.3 Memory model	20
5.4 Defining observation IDs, environment IDs, and constraints	22
5.5 Example problems	23
5.6 Diagnosing the example problems	25
5.7 Sink-side diagnosis generation	26
6 Experimental results	27
6.1 Memory use	27

6.2	Experimental setup	29
6.3	Analysis of a single run	30
6.4	Detection rate	33
6.5	Communications Overhead	35
6.6	Discussion	35
7	Conclusions and future work	39
7.1	Future Work	40
	Bibliography	41

Introduction

A common axiom in computer science is Moore's law, which states that the number of transistors that can be placed on an integrated circuit doubles about every two years. The obvious consequence of this law is the exponential growth in speed, and, more recently, number of processor cores of microprocessors. Looking at Moore's law from a different perspective, integrated circuits with similar performance will also become smaller, less power hungry, and less expensive.

This constant miniaturisation is one of the basic premises of wireless sensor networks (WSNs), in which a large number of low-cost micro-controller boards are connected to a radio, and one or more sensors. These sensor nodes can be powered by a battery, solar cells, or some other off-grid power supply, which allows them to operate in a completely wireless manner. In temperate climates, the power supply is usually provided by a battery, as solar power is not always in steady supply. This brings with it the challenge of longevity of the network. The radio is the main power drain for a sensor node, followed by the processor.

To enable a longer network lifetime, sensor nodes can use the fact that activity comes in bursts related to the sampling period of the sensor. During the periods of inactivity between such bursts, the radio can be disabled, and the processor can go into a low-power sleep mode. As such, for a long-living sensor network, long periods in sleep mode, interspersed with brief bursts of activity are needed.

Sensor networks have been deployed in a wide number of environments, which can often be hostile or hard to reach. Examples include a deployment on Great Duck Island [11], which can only be visited at certain times of the year, and a deployment on an active volcano [10]. Once a node has been deployed, it is often difficult to even detect faults, let alone find their source. A good example is given by Tolle and Culler [16], who describe the deployment of 80 sensors in two redwood trees. The nodes in one tree never returned any data, while 15% of the other tree's nodes also failed to produce any data. This was only detected a month into the project. Langendoen et al. [8] further describe the failures that occurred during a deployment in a potato field. Choi et al. [2] also observed that data yields can be severely affected by faults.

Unfortunately, the very nature of wireless sensor networks, with cheap, simple hardware, wireless transmission, and ad-hoc networking, creates an environment

where faults are often easily introduced, and hard to detect. Some research has been done into achieving more dependability in WSNs, some of which is summarised by Paradis et al. [12], and in the literature study preceding this thesis.

In the wider field of dependability in computer systems, Avizienis et al. [1] provide a taxonomy, which includes four major categories of means to attain dependability: fault prevention, fault tolerance, fault removal and fault forecasting. The latter two are mostly related to measuring and checking whether a system meets its specifications. The former two categories often require a means of detecting faults, and finding their source, which is where fault diagnosis comes in.

When diagnosing a fault, it must first be detected, following which the source of the fault must be localised. There are a number of ways in which this can be done, including the use of heuristics, spectrum-based diagnosis, and model-based diagnosis, more on which can be found in Chapter 2. This thesis investigates applying the technique of model-based diagnosis to wireless sensor networks.

1.1 Problem statement

The main goal of this thesis is to investigate the viability of a model-based diagnoser for wireless sensor networks. Since model-based diagnosis is a general diagnosis technique, it would allow a single diagnoser to find multiple faults of different types. Such a diagnoser has the potential to greatly simplify the process of diagnosing sensor networks.

A diagnoser for WSNs must not only produce correct output, but also work under the limitations imposed by a WSN: power consumption must be kept to a bare minimum, and it must work on the low-end hardware used by the sensor nodes. Aside from these limitations, there are also a number of features that can be taken advantage of in WSNs. The distributed nature of a wireless sensor network allows a computation to be split up into multiple parts, potentially reducing the complexity of the computation that needs to be done at every node.

Further, different sensor nodes provide different views of the network. By combining the information that these separate nodes can gather, a bigger picture can emerge. One could say that the sensor network can be used to detect not only features of the outside world, but also faults that come from within the network.

1.2 Organisation

This thesis is organised as follows. Chapter 2 contains some further background information on wireless sensor networks and diagnosis. Chapter 3 looks at model-based diagnosis, followed by a design for a model-based diagnosis framework in Chapter 4. Next, a proof-of-concept implementation is discussed in Chapter 5, with results from this framework discussed in Chapter 6. Finally, conclusions and future work are presented in Chapter 7.

Background

This chapter provides some background information on the topics discussed in this thesis. Section 2.1 provides further background in wireless sensor networks, while Section 2.2 provides a global introduction to the domain of diagnosis. Section 2.3 looks at a few examples of diagnosis in WSNs, and discusses what diagnosis techniques might be useful for WSNs.

2.1 Wireless sensor networks

As stated in the introduction, wireless sensor networks produce some unique challenges. Not only must power be conserved, data must also be sent back to one or more base stations, also known as sink nodes. Since radio range is limited, multiple hops must sometimes be made. A convenient way of achieving this is by allowing the nodes to self organise into a tree structure with the sink node at its root, as shown in Figure 2.1.

The radio range on sensor nodes is often limited, and unpredictable. Indoors, signals reflecting off walls, ceilings, and objects in a room can cause the same signal to arrive at a receiver multiple times at different signal strengths (or RSSI, received

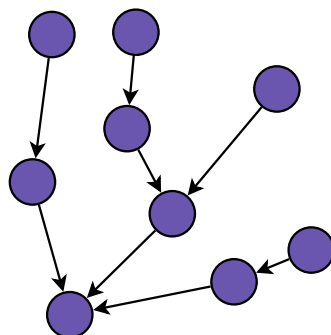


Figure 2.1: An example network, with arrows representing data flow

signal strength indicator levels). Meanwhile background noise levels can vary a great deal. All this can lead to cases where two nodes placed far apart can hear each other well, while nodes that are close together might have great difficulty communicating.

Despite these problems, RSSI is often used as a cheap, low-power way of finding nodes that are geographically close together. Although using RSSI is not extremely effective over shorter distances, the alternatives, including GPS systems and ultrasound are expensive, and cost more battery power. This is the reason why the example applications in Section 5.5 use signal strength to find neighbouring nodes.

Another important restriction is in node hardware. Nodes have limited memory and processing power, and thus large data structures cannot be maintained. Further, the less the processor is used, the better, as this also costs power, and therefore reduces the lifetime of the network. For example, the sensor node used in the Embedded Software group's testbed, the TNode, has 4KB of RAM, 128KB flash memory, and an 8-bit ATmega 128L CPU, running at 8MHz.

2.2 Diagnosis techniques

In general, diagnosis can be seen as finding the root cause of a failure. Here, a failure is defined as a difference between observed behaviour in the system, and expected behaviour. The cause of such a failure can be one or more faulty components in the system, which is what the diagnosis process tries to find. Avizienis et al. [1] have created a taxonomy of dependable computing, which this thesis will try to adhere to as much as possible.

There are a number of ways of performing fault diagnosis, three of which will be highlighted in this section: heuristics, spectrum-based fault localisation, and model-based diagnosis. An overview of the latter two techniques can also be found in Zoetewij et al.[17].

In the heuristics-based approach, as the name implies, the diagnoser would reason based on a number of rules of thumb, and choose the most likely root cause of a failure. An example would be an expert system, where the details of the failures are matched to a set of rules, and the most likely root cause is identified.

Spectrum-based fault localisation is based on the idea that given a known input, and expected output of a system, by running a trace of which components of the system have been used in producing said output, one can identify which components have a large chance of being faulty or are working correctly.

Example 2.2.1 (Spectrum-based fault localisation). Given a test set of a software system, and the ability to trace the execution path of the software, the code of the software system can be exercised. Any test cases that fail would have the code blocks on their execution path marked as being part of a failure. Any test cases that succeed would have their components marked as being part of a successful run. The resulting table of successes and failures of code blocks can then be used to find the code blocks most likely to contain the fault.

Model-based diagnosis is a technique where a model of a system is created, which is subsequently compared to observations from the actual system. Based on predictions made from the observations, and which components were involved in making those predictions, diagnoses can be made. A more in-depth view of model-based diagnosis will be presented in Chapter 3.

2.3 Diagnosis in WSNs

Currently, most diagnosis systems in WSNs are aimed at diagnosing specific faults, such as detection of crashed nodes, sensor faults, or identifying faulty behaviour in nodes. A number of these systems can be found in Paradis et al. [12] and in the preceding literature study.

There are a few larger systems, such as Sympathy [13], which uses a centralised diagnosis engine on the sink node. The problem here lies in the fact that a large number of observations need to be sent to the sink node, increasing communications overhead. This problem is partially solved by using overhearing of nodes close to the sink node, but further nodes will still need to communicate.

Looking outside of the field of WSNs, Kurien et al. [7] suggest a system for networked embedded components, where a network of components is split into multiple parts, each of which has one or more model-based diagnosers. These diagnosers exchange observations and predictions to continually refine their local diagnoses, until no further improvement can be made. The communication load in this system is still quite high, however, since a number of messages are exchanged every iteration, until a diagnosis is found.

Kalech et al.[6] look at the diagnosis of multi-agent systems, specifically at coordination failures. Their diagnoser observes the actions taken by agents in a system, and attempts to find candidate abnormal agents, using model-based diagnosis techniques. This problem is very similar to finding nodes that are behaving abnormally, however, the algorithms used are not always suitable to the hardware restrictions found in wireless sensor networks.

Looking back at the techniques discussed in Section 2.2, and what can be applied to WSNs, the heuristic techniques can only diagnose failures that were anticipated by the developer. Thus, if unexpected faults crop up, such a system might make a mistake, or be unable to produce a diagnosis.

Spectrum-based fault localisation offers a solution for a more general diagnostic engine, but it has the disadvantage of needing an oracle to give the expected output from the known set of inputs. Further, the communications requirements of running a centralised system would be quite large, as input and output values would have to be sent to the sink. Distributing the system would also be difficult, as the power of spectrum based diagnosis comes from combining a large number of execution paths.

This leaves model-based diagnosis as an option, but here, a centralised diagnoser would require a large number of observations to be sent to the central sink. The communications cost of such a system would be high. Distributing a system is however a possibility, since many of the problems encountered in wireless sensor networks

can be modelled locally, at the node level. Such a distributed system where every node runs its own local model of its environment is a potential candidate for a more general diagnoser, and is what is explored in this thesis.

Model-based diagnosis

Model-based diagnosis traces its roots back to the 1980's, with the search for a more general theory of diagnosis. The foundation was laid by de Kleer et al. [3], and Reiter [14]. The basic premise is that by combining a model of a system and a number of observations from the system, one can obtain a diagnosis. This section aims to provide a basic primer on model-based diagnosis and provide the background necessary for the rest of this thesis. For a more complete and formal approach, please refer to de Kleer and Reiter's work.

3.1 The Model

Some terminology and notation must be introduced to help explain how model-based diagnosis works. The notation used here comes primarily from Reiter [14]. Model-based diagnosis works by checking the consistency of a system consisting of a system description (SD), a finite set of components (COMPS), and a number of observations (OBS) to produce a diagnosis.

The system description is a set of logical statements, or constraints, of how components in the system behave, and how they are connected. The predicate AB is used to indicate abnormal behaviour, so $\neg AB(X) \rightarrow P$ means if component X does not behave abnormally, then P must apply.

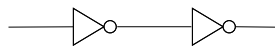


Figure 3.1: Two cascaded inverters

Example 3.1.1. A system of two cascaded inverters, could be described by

$$\begin{aligned} \neg AB(INV_1) &\rightarrow output(INV_1) = not(input(INV_1)) \\ \neg AB(INV_2) &\rightarrow output(INV_2) = not(input(INV_2)) \\ input(INV_2) &= output(INV_1) \end{aligned}$$

Next, a set of observations that were made can be added, describing measurements made on the system. For example, if the input of the system in Example 3.1.1 is observed to be 1, and the output 0, the observations would be $input(INV_1) = 1$ and $output(INV_2) = 0$.

Using the terminology introduced at the beginning of this paragraph, Reiter gives his definition of diagnosis:

Definition 3.1.1. A diagnosis for $(SD, COMPS, OBS)$ is a minimal set $\Delta \subseteq COMPS$, such that

$$SD \cup OBS \cup \{AB(c) | c \in \Delta\} \cup \{\neg AB(c) | c \in COMPS - \Delta\}$$

is consistent.

Thus, a diagnosis is a minimal set of components which must be behaving abnormally to make the system $SD \cup OBS$ consistent. Minimal diagnoses are referred to using the notation $\Delta = [c_1, c_2, \dots]$

Example 3.1.2. This example is based on the system in Example 3.1.1, with observations $input(INV_1) = 1$ and $output(INV_2) = 0$. Since these observations are inconsistent with the system description, the diagnosis is not the empty diagnosis $[\]$. Since our problem is relatively simple, there is a trivial way of finding a solution, which consists of generating every consistent diagnosis, with the smallest sets first. Doing this we obtain $\Delta_1 = [INV_1]$ and $\Delta_2 = [INV_2]$. $\Delta_3 = [INV_1, INV_2]$ is not a minimal diagnosis, since it is a superset of Δ_1 and Δ_2 .

3.2 Finding a diagnosis

This section presents a new, slightly more complicated running example to demonstrate the techniques used to perform model-based diagnosis. The example uses an existing model of a WSN to check whether a certain sensor node is producing abnormal data. The model could be based on localisation data obtained through GPS or beaconing, or manually inputted on network creation.

Example 3.2.1. Take the model in Figure 3.2, where groups of nodes that should have similar sensor values have been coloured. A model could be created for the group A,B,C as follows:

$$\begin{aligned} \neg AB(A) \wedge \neg AB(B) &\rightarrow sensor(A) \approx sensor(B) \\ \neg AB(A) \wedge \neg AB(C) &\rightarrow sensor(A) \approx sensor(C) \\ \neg AB(B) \wedge \neg AB(C) &\rightarrow sensor(B) \approx sensor(C) \end{aligned}$$

Here, the \approx operator specifies that the sensor values ($sensor(x)$) must return similar values. Connections between components could be worked out based on the figure. Now, suppose that $sensor(A)$ is measured to be 10. The model will still

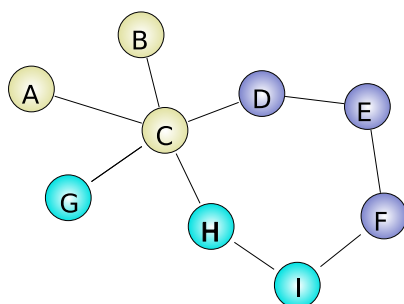


Figure 3.2: Localised model of a WSN. Members of each coloured group should have similar sensor values.

be consistent, and the diagnosis is that there are no faulty components. Next, if we measure $sensor(B) = 15$, $sensor(A) \approx sensor(B)$ does not hold, and thus the diagnoses will be that either A or B is faulty. Finally, adding $sensor(C) = 10$, to make the above equations consistent, either A and C must both be faulty, or B must be faulty. The diagnoses are thus $[A, C]$ and $[B]$

In the above example, the minimal diagnosis is found intuitively, and can easily be found by generating consistent diagnoses until no new set can be found. This is, however, an inefficient method if a larger system is being diagnosed.

3.2.1 Conflict Sets

Reiter [14] and de Kleer [3] propose different methods to find the set of all diagnoses, by computing minimal conflict sets. Using Reiter's definition,

Definition 3.2.1. A conflict set is a set $\{c_1, \dots, c_k\}$ of components that would make the system $SD \cup OBS \cup \{\neg AB(c_1), \dots, \neg AB(c_k)\}$ inconsistent. The conflict set is minimal, if it has no proper subset that is also a conflict set.

In other words, at least one of the elements of a conflict set is functioning abnormally. The set of all diagnoses for the system can then be found by calculating the minimal hitting sets of the minimal conflict sets of the system.

Example 3.2.2. Using the model from Example 3.2.1, and the observations $sensor(A) = 10$ and $sensor(B) = 15$, $\langle A, B \rangle$ is a conflict set, as either A or B must have a broken sensor. Introducing the observation $sensor(C) = 10$ introduces another minimal conflict set $\langle B, C \rangle$. The minimal hitting sets for $\langle A, B \rangle$ and $\langle B, C \rangle$ are then $[B]$, $[A, C]$. This is illustrated in Figure 3.3.

3.2.2 Reiter's Algorithm

Reiter bases his algorithm on the generation of a tree, called the HS-tree (for Hitting Set). This is a tree with nodes labelled by conflict sets, or \checkmark if it has no label. If

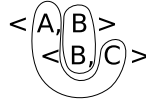
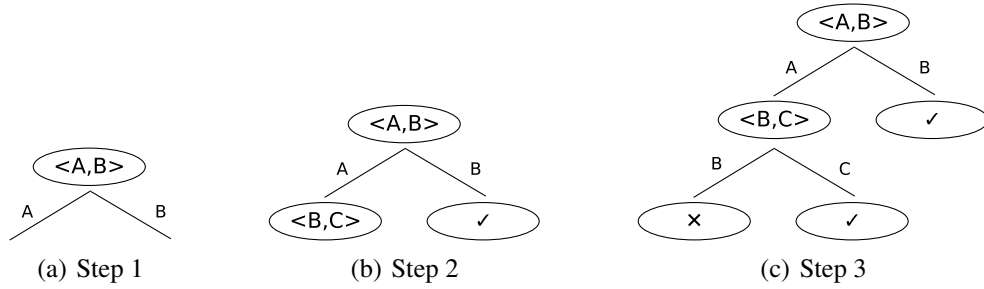
Figure 3.3: Minimal hitting set of the conflict sets $\langle A, B \rangle$ and $\langle B, C \rangle$ 

Figure 3.4: Building up an HS Tree for example 3.2.3

a node has a label, it has an edge going out for each element of the conflict set the node is labelled with. These edges are labelled by this element. A child node must then be labelled with a conflict set, that does not contain any elements of the set of edge labels going from that child node to the root node. If such a conflict set does not exist, it is labelled by \checkmark .

Reiter's method is based on the fact that such a tree contains all of the minimal hitting sets for the conflict sets. By the above definition, it may contain more of such sets, but by pruning unneeded nodes during generation of the tree (these nodes are marked by \times), only the minimal tree remains, and the set of diagnoses can be read from it.

The remaining question is where the conflict sets used to label the nodes of the tree come from. Reiter suggests to either reuse a previously generated conflict, or use a theorem prover to generate a conflict based on the system $SD \cup OBS \cup COMPS - H(n)$ where $H(n)$ is the set of edge labels on the path up to the root. The implementation of such a theorem prover is application specific, but could be a solver.

Example 3.2.3. (Reiter's Algorithm). Returning to the running example for this section (Example 3.2.1, the creation of the tree is shown in Figure 3.4. In Figure 3.4(a), a root node is created, and a label is taken from our theorem prover, in this case $\langle A, B \rangle$. Edges are created, labelled A and B.

In Figure 3.4(b), the child at the end of edge B provides a consistent set, so is labelled with \checkmark . The child on edge A is not yet consistent, however, so a new conflict set is generated, based on the system $SD \cup OBS \cup \neg AB(A)$ labelling it $\langle B, C \rangle$.

In Figure 3.4(c), edges are generated for B and C, with C again producing a consistent system. B is pruned, since it has already occurred higher in the tree. Reading

the complete tree from \checkmark to root provides us with two minimal hitting sets: $[B]$ and $[A, C]$.

3.2.3 De Kleer's algorithm: GDE

De Kleer [3] has designed the General Diagnostic Engine (GDE), which uses an inference engine to calculate predictions based on incoming observations. During the inference process, the constraints that the engine uses to infer a result are stored with the result as an environment. If two observations or predictions then clash, a symptom has been found. The environment of such a symptom can then be used as the conflict set. As an optimisation, any environment that has produced a symptom is not explored further, since any conflict sets generated by such an environment would not be minimal (they are always a proper superset of the original environment).

Once all conflict sets are found based on current observations, the minimum hitting set is found, and the minimal diagnoses are known.

Example 3.2.4 (GDE). Running the inference engine on Example 3.2.1, with an observation of $sensor(A) = 10$ would produce the predictions $sensor(B) = 10$ and $sensor(C) = 10$, with environments $\{A, B\}$ and $\{A, C\}$ respectively. Now, if an observation $sensor(B) = 15$ were to enter the system, a symptom would be found between this new observation and the prediction $sensor(B) = 10$, $\{A, B\}$, and the conflict set $\langle A, B \rangle$ would be produced. The resulting diagnoses would be $[A], [B]$. Finally, if $sensor(C)$ was observed to produce the value 10, a second conflict would be added to the existing conflict set, $\langle B, C \rangle$. Taking the minimum hitting set of both conflicts results in the diagnoses $[B], [A, C]$.

Design of a model-based diagnosis framework for WSNs

The primary requirement of a diagnoser for wireless sensor networks is that it should use as little power as possible. This translates into a design that attempts to minimise communications first, and minimise processor use second. Another issue that must be kept in mind, is that sensor nodes have very limited resources: the TNode nodes used in this project have 4KB of RAM, and 128KB of program memory.

Both Reiter's algorithm and de Kleer's GDE, as discussed in the previous chapter, do not map directly onto such a platform. First of all, they are both centralised algorithms, meaning that all observations must be sent to a central diagnoser. Reiter's algorithm runs into further problems because it requires a solver oracle to provide it with labels for its HS-tree. Implementing such a solver efficiently could prove to be prohibitive with the limited memory available on a node. De Kleer's GDE has the problem that it uses an assumption-based truth maintenance system (ATMS) in its inference engine, which would quickly use up resources. Further, calculating the minimal hitting set is an NP-complete problem.

The basic premise of the solution proposed in this thesis is to use the distributed nature of a WSN to solve these problems. By running a simple inference engine on every node, with a small local model, nodes can generate conflict sets. If the model can be made to only use local observations, observations no longer need to be sent over the network. As a final step, conflicts generated by the local inference engine can be sent to the sink node, which then calculates the minimal hitting set of these conflicts to produce the diagnoses.

This proposed solution has a number of advantages. First of all, packets only need to be transmitted when reporting conflict sets to the sink. This only occurs when a symptom is actually detected, so during normal operation, no messages would be sent. Secondly, processing is distributed over all the sensor nodes, reducing the load on a single node.

This chapter describes a design of a model-based diagnoser based on the basic premises laid out in the previous paragraphs. Section 4.1 looks at data-structures

for the observations, while Section 4.2 looks at the model definition. Section 4.3 then introduces the node-level inference engine design, which produces conflicts that can be sent to the sink. Finally, the section describes the sink-side diagnoser, which produces the actual conflicts.

4.1 Observations, predictions and environments

An incoming observation is what triggers the inference engine to start doing work. Observations have an identifier (ID), which refers to what has been measured, and a value, which can be either a numerical value or a Boolean variable. For example, an observation could refer to a sensor value that has been read, or the fact that a node did not receive a reply from a neighbouring node.

An observation has no supporting environment. They are just measurements taken from the system, and as such, are not based on any assumptions. Predictions on the other hand, have a supporting environment, based on the environments of the constraints that generated the prediction.

Both observations and predictions can be stored in the same database, and differentiated by whether or not they have an environment. They are in essence a list in the form $obsid, value, envid_1, envid_2, \dots, envid_n$.

4.2 Model definition

Every node has its own local model of how the world should behave from its point of view. For example, a node could have constraints that state that its neighbouring nodes should be producing similar sensor values, or that when it sends a message to a node, an acknowledgement is expected.

Initially, on model definition, constraints would be abstract, for example of the form:

$$\neg AB(node) \wedge \neg AB(n_i) \rightarrow sensor(node) \approx sensor(n_i) \\ |n_i \in NEIGHBOURS(node)$$

Here, the predicate $sensor(x)$ refers to the sensor value from x . This model therefore states that given that the $node$ and its neighbours n_i are behaving normally, the $node$'s sensor values should be approximately the same as the sensor values of its neighbours. Note that some implicit assumptions were made in defining this constraint. For example, there is an assumption that node values will change at a low enough rate, that even if a node has an older observation from its neighbours, the constraint will still be valid. A further discussion on these assumptions will follow in Section 5.5.

To translate this abstract model to a form usable by the inference engine, the initial part of the equation would become the supporting environment for the constraint. i.e. $\neg AB(node) \wedge \neg AB(n_i)$ becomes $env(node)$, and $env(n_i)$. The last part of the constraint would need to be split into the observation IDs $sensor(node)$

and $sensor(n_i)$. Finally, such a constraint is assigned a constraint type identifier.

To get the actual constraints from such an abstract model, the model needs to be instantiated on the node, replacing the abstract $node$ and n_i with actual data. Further, constraints need to be generated for every neighbour. Data on which node the model is running on, and which neighbours exist is only available at run-time in an ad-hoc network, so this final step must be done on the node.

Constraints could thus be conceived as being of the form $constrainttype, obsid_1, obsid_2, \dots, obsid_n, envid_1, envid_2, \dots, envid_m$. Here, the constraint type will be predefined, and observation IDs will be generated dynamically at run-time. The environment will also need to be dynamically generated.

4.3 Node-level inference engine

The goal of the inference engine is to take incoming observations, make predictions based on these observations, and generate conflicts when a symptom is found between two observations and predictions. The inference engine will run on the nodes and send conflicts back to the sink. The component will need two main entry points. Initially, a model needs to be instantiated by adding constraints to the engine. This could be after boot of the node, or when the list of neighbouring nodes changes. The second entry point is the addition of observations. The following example demonstrates the basic inference process.

Example 4.3.1. The basic inference process.

The following abstract model:

$$\neg AB(node) \wedge \neg AB(n_i) \rightarrow sensor(node) \approx sensor(n_i) \\ |n_i \in NEIGHBOURS(node)$$

would be translated into a constraint with type $constraint_implication$, observation IDs $obsid_{sensor}(node)$ and $obsid_{sensor}(n_i)$, and environment IDs $envid_{sensor}(node)$ and $envid_{sensor}(n_i)$ on model definition.

At run time, the model will be instantiated. If node A has neighbours B, C and D, the following constraints will be generated on node A, with supporting environments between curly brackets:

$$sensor(A) \approx sensor(B), \{A, B\} \\ sensor(A) \approx sensor(C), \{A, C\} \\ sensor(A) \approx sensor(D), \{A, D\}$$

On reading its own sensor value as 10, A will have an observation $sensor(A) = 10$. This triggers all three constraints, and creates the predictions (again with sup-

Algorithm 4.3.1: *addObservation(observation)*

```

if observationDB contains observation.id then
  if existingObservation.value  $\neq$  observation.value then
    purge existing observation and any predictions and symptoms based on it
  else return
  end if
end if
processObservation(observation)

```

porting environments in curly brackets):

$$sensor(B) \approx 10, \{A, B\}$$

$$sensor(C) \approx 10, \{A, C\}$$

$$sensor(D) \approx 10, \{A, D\}$$

Now, if node A receives a notification that B's sensor has read 15, A adds the observation $sensor(B) = 15$. The inference engine will immediately signal a symptom with $sensor(B) \approx 10, \{A, B\}$, from which a minimal conflict $\langle A, B \rangle$ can be generated and sent to the base station.

As stated earlier, the second entry point to the inference engine is the addition of an observation to the engine. Initially, this would check whether an observation already exists in the database of existing observations, and either replace it in the case where it already exists but has a different value, or return if the value of the existing observation is the same, as can be seen in Algorithm 4.3.1. If an observation is replaced, all predictions and previously detected symptoms that were based on the observation are purged too. After the observation database has been cleaned the inference starts making predictions.

The algorithm used to make predictions is *processObservation()*, for which pseudo-code is given in Algorithm 4.3.2. It is a recursive algorithm. Initially, it obtains predictions for a given observation, and checks whether a symptom is produced between these new predictions, and existing observations or predictions. Next, it checks whether the new prediction already exists in the prediction database. If it does exist, and the new prediction's environment is a superset of an existing prediction, or the same, the new prediction is not processed any further. If the new prediction already exists, but the existing prediction's environment is a superset of the new prediction's environment, the existing prediction is purged in favour of the new prediction. *processObservation()* is then recursively called on the new prediction.

Execution of a constraint is a potential problem. Complex constraints would require a truth maintenance system (TMS) to perform the inference. If kept simple, however, heuristics might be used. For example, a constraint of the form $\neg x \vee y$ could be translated to if $x = \text{true}$, then $y = \text{true}$, and if $y = \text{false}$, then $x = \text{false}$. In

Algorithm 4.3.2: *processObservation(observation)*

```

for all constraint in model do
  if constraint.containsObservation(observation) then
    prediction  $\leftarrow$  constraint.execute()
    if  $\neg$ prediction then
      continue to next constraint
    end if
    noSymptoms  $\leftarrow$  true
    for all existingObs in observationDB do
      if existingObs.id = prediction.id then
        if existingObs.value  $\neq$  prediction.value then
          signal symptom between existingObs and prediction
          noSymptoms  $\leftarrow$  false
        else if prediction.env  $\supseteq$  existingObs.env then
          continue to next constraint
        else if existingObs.env  $\supset$  prediction.env then
          observationDB.remove(existingObs)
        end if
      end if
    end for
    if noSymptoms then
      processObservation(prediction)
    end if
  end if
end for

```

other cases, no inferences can be made. By including only the heuristics needed to make predictions on the types of constraints in the model, the system can be kept simple.

On successful inference, the union of the constraint's supporting environment and the supporting environment of the predictions used in the inference is computed, and a new prediction is created, labelled with this new supporting environment.

4.4 Creating a diagnosis

The previous section describes a method for the identification of minimal conflicts for a local model. The next step is the actual generation of the diagnosis. This involves calculating the minimal hitting set, which is an NP-complete problem.

The calculation of diagnoses can be done incrementally, using a brute force approach. Given a single minimal conflict, for example $\langle A, B, C \rangle$, The set of diagnoses would be [A], [B], [C]. Now, given a second minimal conflict, $\langle A, B, D \rangle$, diagnoses [A] and [B] are contained in both conflicts, and can remain. Diagnosis

[C] however is not, and can be eliminated. The super-sets that cover the second conflict must be included however: [A,C], [B,C], [D,C]. The first two are covered by [A] and [B], however, and are therefore not needed, leaving the minimal diagnosis as [A],[B],[C,D]. Adding a third conflict $\langle A, B, E \rangle$ again leaves us with [A], [B]. [C,D] is replaced by just its superset, [C,D,E].

4.5 Discussion

A major drawback of the above system is the lack of support for a more general logical language. Since we are trying to avoid the use of an actual truth maintenance system, the types of constraints that can be processed depend on the heuristics implemented. This inherently specialises the system to the types of constraints that can be defined, but greatly increases the simplicity of the system.

A further problem is the availability of observations. In some situations, constraints will only be triggered by the reception of external observations. In the example in the previous section, the value of B's sensor was necessary. There are two potential solutions to this problem: hope that the value will pass through our node on the way to the sink, or deliberately spread potentially useful observations to other nodes. The latter could perhaps be achieved through the piggy-backing of selected observations on routine messages.

Lastly, lack of a complete picture might hurt the quality of the diagnoses. As no node will have a full model of the network, certain inferences might not be made, resulting in a failure to find larger-scale conflicts.

Implementation

This chapter deals with details of the implementation of a proof of concept model-based diagnoser, as described in the previous chapters. The implementation of the node-side inference engine was done in TinyOS, an operating system that is often used for sensor networks. In total the inference engine has about 1700 lines of code, including comments. The test application adds another 1250 lines, and the sink-side client adds 2000 lines of Java code, including a graphical user interface.

Section 5.1 gives a brief introduction to TinyOS, and some of its concepts, followed by Section 5.2, which explains how observations are identified. Section 5.3 gives details on how memory management is handled for the observation and model database. Section 5.4 then explains how new constraints, observation IDs, and environment IDs can be added to the system. Section 5.5 introduces some example problems that will be used to test the diagnoser's performance, using the application described in Section 5.6. Finally, the implementation of the sink-side application that generates the diagnoses from conflicts it receives will be described in Section 5.7.

5.1 TinyOS

TinyOS [9] is an operating system written specifically for sensor networks. It uses its own derivative of C, nesC [4], which adds a number of concepts to the C language. A nesC application consists of a number of components, which are wired together in a configuration. Components communicate amongst each other using bidirectional interfaces, which consist of commands that allow the user to call a component, and events which feed information back to the user.

TinyOS provides a number of standard components, such as timers, LEDs, and some communication protocols. It works on a number of different platforms, including telos, micaz, and the TNode, the platform used in the Embedded Software group's own sensor network testbed.

As an example of a configuration, see Figure 5.1, which shows the different components in the inference engine, and their wirings.

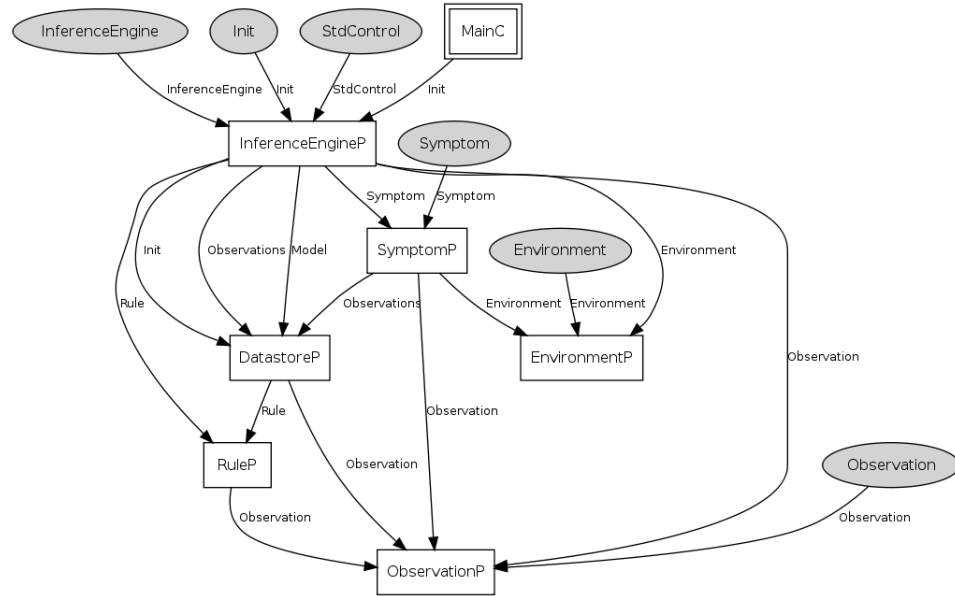


Figure 5.1: Component structure of the inference engine. Ovals represent interfaces, squares represent components, and the arrow labels show which interface is called.

5.2 Observation and environment identifiers

As discussed in Section 4.1, every observation and environment needs a unique identifier, which can be used when referring to it. To conserve memory, two different identifiers are used in the actual inference engine implementation.

The first identifier is a global, network-wide identifier. It is variable sized, and consists of an observation or environment type, and some parameters. For example, an observation identifier for a message sent from *A* to *B* would consist of a single byte type *OBS_TYPE_SEND*, with parameters *A* and *B*, each consisting of a two byte node address.

The second identifier is used locally, and consists of an 8-bit observation ID. The mapping between the global observation identifier and the local identifier is made when adding an observation to the observation/prediction database.

The advantage of using two separate identifiers is that less space is needed when an identifier is used multiple times. For example, if the above observation identifier is used in two different constraints, using the global identifier would cost 10 bytes, while using local identifiers would only cost 7 bytes of RAM. The same applies when an observation or prediction gets added to the inference engine's database.

5.3 Memory model

The size of many of the data structures used by the inference engine are variable. For example, the length of a constraint depends on the constraint type. For predictions,

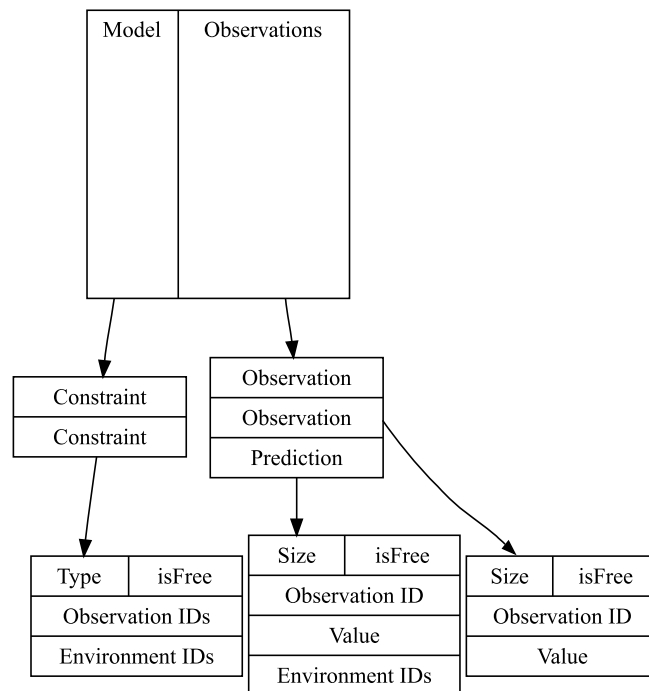


Figure 5.2: Data structure used to store the model, observations, and predictions. The model contains constraints, with the structure indicated at the bottom left. The observation database contains observations and predictions, as detailed on the bottom right and center, respectively.

the problem is exacerbated. Not only can the size of their value (e.g. Booleans, 16-bit integers) differ, but they also have an unpredictable supporting environment size.

Unfortunately, TinyOS does not support dynamic memory, and the results of a malloc call are not always correct. To still enable efficient use of memory, a custom memory manager was implemented for the model, observation and ID databases. The main model/observation database consists of two parts: the model occupies the first part of the reserved memory block, and the observation/prediction database occupies the second part. An overview of the data structures used can be found in Figure 5.2.

The model consists of a series of constraints, separated by a single byte header containing the constraint type, and whether the constraint is valid or not. This field is currently not used, as a model is currently static, but is included to allow future support of dynamic models. The last constraint is marked by an empty header. This way, constraints are stored in a compact way.

The observations and predictions follow immediately after the model in memory. They are marked with a single byte header containing the total size of the observation/prediction, and whether it is still valid or not. Predictions also contain environment IDs, which refer to their supporting environment.

Since the observations can be removed, and their size is dynamic, the observation/prediction database needs to be compacted regularly, to free up space for new observations. This is done after an observation has been added, and the inference process is complete, by moving any observations or predictions that are still valid closer to the top of the observation store, overwriting any invalid observations in the process.

Aside from the main model/observation database, there are three other databases, the observation and environment ID databases described in Section 5.2, and a database containing all of the symptoms found by the inference engine. This symptom database contains a header, and a reference to two conflicting observations and/or predictions. The header consists of a single byte, the last 2 bits of which define whether the symptom has or has not been sent, whether it has been invalidated, or whether it is free. Once a symptom has been sent, the first 6 bits of the header define a symptom ID. This ID is used to send an “invalid symptom” packet to the sink when one of the observations supporting the symptom is removed by the inference engine.

The symptom database has one problem, however: if an observation is moved in memory, the pointer to its location is no longer valid. This is solved by having the observation database signal an “observation moved” event, allowing the symptom database to update its references.

Note that sending out symptoms is susceptible to packet loss, which can cause a conflict to be missed by the sink when sent over an unreliable transport. The choice of how to send symptoms is left to the developer. The proof of concept implementation uses CTP, the same protocol that is used to send back sensor data.

5.4 Defining observation IDs, environment IDs, and constraints

Currently, observation IDs, environment IDs and constraints are manually defined, but they are organised in such a way that automatic generation could be done in the future. New definitions are relatively easy to add, with just a few small changes necessary to some header files, and a small piece of extra code needed for constraints. This allows rapid deployment of new constraint types.

Environment IDs are defined by an enum *environment.type*, and an array containing the size of the parameters of each environment type. For example, a sensor environment has type *ENV_TYPE_SENSOR*, and a two byte *nodeId* parameter defining which node the sensor is on.

Observations are defined by a similar *observation.type*, the size of the observation’s parameters, and the size of its value (e.g 2 bytes for a 16-bit integer). Optionally, the observation component’s equals command can be modified. This was, for example, done to deal with approximate sensor values. For an example of an observation definition, see Example 5.4.1 below.

Finally, constraints are again defined by a *constraint.type*, and the number of observation and environment IDs contained in the constraint. Further, an execute

method must be defined for a constraint, which, given a constraint and the observation that triggered the constraint, can return a prediction.

Example 5.4.1. Two observation types, send and receive are defined as follows:

```
enum {
    OBS_TYPE_SEND = 1,
    OBS_TYPE_RECEIVE,
};

typedef struct obs_params_send_recv {
    am_addr_t sender;
    am_addr_t receiver;
} obs_params_send_recv_t;

typedef struct obs_info {
    size_t params_size;
    size_t value_size;
} obs_info_t;

obs_info_t const obs_info[2] = {
    {sizeof(obs_params_send_recv_t), sizeof(bool)},
    {sizeof(obs_params_send_recv_t), sizeof(bool)},
};
```

5.5 Example problems

This section will introduce two example applications for the diagnoser, detection of bad sensor values, and detection of crashed nodes. These examples both rely on the concept of neighbours. In the ideal case, neighbours would be localised using techniques such as GPS. However, such hardware is expensive, and costs power to operate. In the proof of concept application, the list of neighbouring nodes is therefore approximated by using the strength of the received radio signal.

5.5.1 Detecting bad sensor values

This first example is based on the idea that nodes that are close together should produce similar sensor values. A network consisting of a number of nodes produce sensor values at regular intervals, and send them to the central sink node. A further assumption that is made is that sensor values only change gradually over time. If sensor values change too quickly, a node's sensor value might differ too greatly from a previously observed neighbour's sensor value, and a symptom might be found.

An abstract model for such a system can be seen below:

$$\neg AB(node) \wedge \neg AB(n_i) \rightarrow sensor(n) \approx sensor(n_i) | n_i \in NEIGHBOURS(node)$$

5.5.2 Detecting crashed nodes

The example given in this section shows a number of constraints that could be used to detect whether a node has probably crashed, or that communications with the node have failed. The idea is that a node can send a periodic ping message to a neighbouring node, and if it does not reply, a conflict is produced. The following models this situation:

$$\begin{aligned} &\neg AB(link(node, n_i)) \rightarrow send(node, n_i) \rightarrow receive(n_i, node) \\ &\neg AB(n_i) \rightarrow receive(n_i, node) \rightarrow send(n_i, node) \\ &\neg AB(link(n_i, node)) \rightarrow send(n_i, node) \rightarrow receive(node, n_i) \\ &|n_i \in NEIGHBOURS(node) \end{aligned}$$

Arguably, this could be collapsed to a single constraint, $\neg AB(link(node, n_i)) \wedge \neg AB(n_i) \wedge \neg AB(link(n_i, node)) \rightarrow send(node, n_i) \rightarrow receive(node, n_i)$, since without communication, the sending node will never know whether node n_i received and processed the original message. However, for the sake of clarity, and to demonstrate the inference process, the three constraint model will be used.

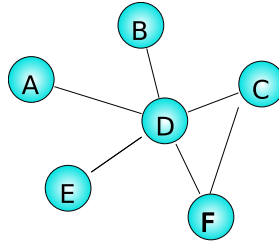


Figure 5.3: Network structure for Example 5.5.1

Example 5.5.1. In Figure 5.3, if C were to ping D, then the inference engine will predict that D receives the message, with supporting environment $\{link(C, D)\}$. In a next step, the inference engine will predict that D will send a reply, now with supporting environment $\{link(C, D), node(D)\}$. Finally, the engine will predict that C would receive the reply, with environment $\{link(C, D), node(D), link(D, C)\}$.

If C detects that it has never received a reply, for example due to a timeout, this new observation will produce a symptom with the prediction stating that we should receive a reply if we send a message. This results in a minimal conflict set $\langle link(C, D), node(D), link(D, C) \rangle$, and minimal diagnoses $[link(C, D)]$, $[node(D)]$, $[link(D, C)]$.

Given that F performs the same regular polling, F would produce a conflict $\langle link(F, D), node(D), link(D, F) \rangle$, if node D is really down. This would result in a new set of minimal diagnoses: $[node(D)]$, $[link(C, D), link(D, F)]$, $[link(C, D), link(F, D)]$, $[link(D, C), link(D, F)]$, $[link(D, C), link(F, D)]$.

On the other hand, if F does not detect a symptom, the diagnosis would not change.

5.6 Diagnosing the example problems

To test the inference engine, a TinyOS application was created. The core of the application is very simple, and based on the multi-hop oscilloscope example source code included with TinyOS. Every node has a fake sensor component, which always generates similar values (hexadecimal `0xbeef ± 2`). The sensor further contains a timer, that if set to go off, will “break” the sensor, and make it produce invalid values (`0xdead ± 2`). This sensor is sampled every ten seconds. Once five samples have been generated, a packet is sent back to the sink using TinyOS’s standard collection tree protocol (CTP).

The second component of the application obtains observations from the main application, and feeds them to the inference engine. This “glue” component also creates the model that is used by the inference engine. The model is generated at run-time, based on CTP’s neighbour list, and the constraints described in Section 5.5. Three observation types are used. Sensor value observations are recorded when sending them, intercepted when forwarding packets, and overheard from the network. Send and receive observations are based on CTP’s regular beacon messages. Only observations relevant to the current model are stored, to conserve memory.

Generation of send and receive observations works differently from what was described in Section 5.5.2. The model described in that section is still used to demonstrate the inference process. However, to prevent unnecessary communications, the application uses CTP’s existing beacon messages. On model instantiation, the initial *send(node, neighbour)* observation is set to true. This implies that neighbouring nodes will always receive a beacon message. The reply *receive(node, neighbour)* is taken from received beacons. The node will record which neighbours a beacon message has been received from during a set interval. Any neighbours that a beacon has not been received from during that interval will have a negative observation recorded, and vice-versa.

In an actual deployment, a much simpler model could be used to model CTP’s beacons. This would not only be a better match for what actually happens, but conserve memory as well, as fewer constraints and predictions would need to be stored. However, to demonstrate the inference process, the more complex model described in Section 5.5.2 was used in the proof of concept application.

Once a symptom has been detected, the inference engine signals a *new conflict* event. The conflict is then sent over the network towards the sink, which then forwards the message to a client connected to the serial port. Conflict invalidation messages are treated similarly.

A number of parameters can be tweaked in the implementation. For the inference engine, the size of the different databases, and how far apart sensor values may be can be adjusted. New constraint types can be added to the inference engine by defining the number of observations needed for a constraint, its number of environments, and writing a function that produces predictions if the constraint fires. New observation and environment types only need new IDs to be defined.

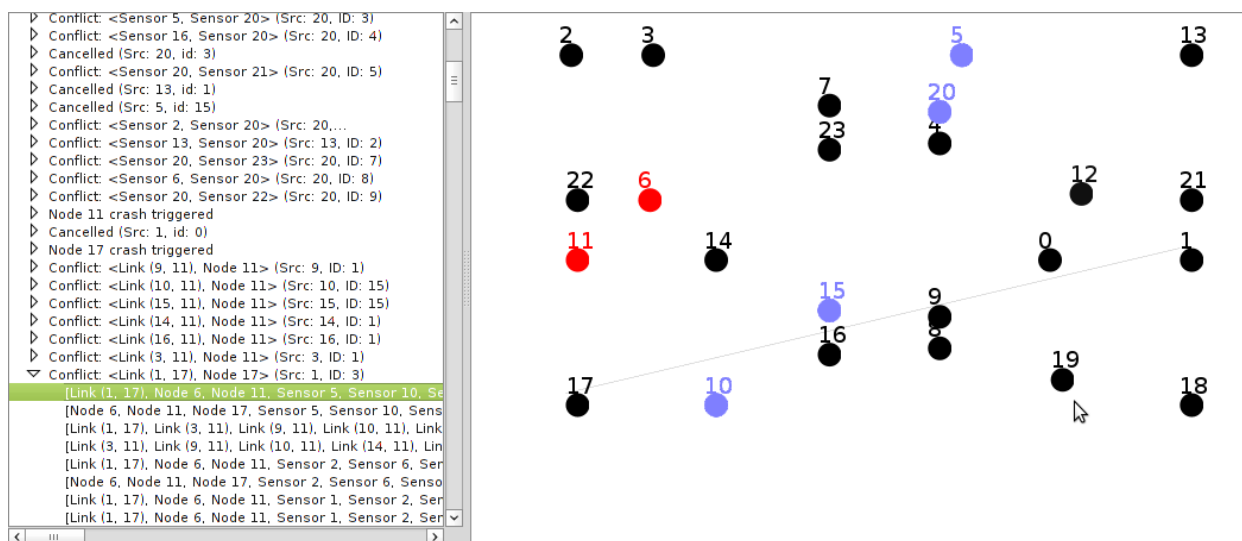


Figure 5.4: A screen-shot of the sink-side diagnoser's GUI. Red nodes have crashed, and blue nodes have sensor failures. The line indicates a link failure.

The glue component allows the user to define whether links with poor reception quality should be filtered from the neighbour list, and a list of which nodes should die, and at what time. Further, sensors can be told to fail on certain nodes at a given time. Lastly, the time at which the model is created can be assigned.

5.7 Sink-side diagnosis generation

When a conflict reaches the computer connected to the sink, it is fed to a listener client written in Java. This client maintains a database of currently active conflicts, and continually generates the set of minimal diagnoses, using the brute force approach described in Section 4.4. The diagnoses are then sorted by their likelihood. The ordering of diagnoses is based on their cardinality. Given diagnoses of equal length, diagnoses with more link failures are given priority.

The diagnoses are currently output as XML and to a GUI interface, which can be seen in Figure 5.4. The left-hand side provides a list of all the conflicts that occurred, with the calculated diagnoses after that event. The right hand side shows sensor failures in blue, node failures in red, and any link failures as lines.

Experimental results

Using the framework described in the previous chapters, a number of experiments were performed to verify the diagnoser’s accuracy, and to measure its execution cost. The experiments were initially performed in TinyOS’s simulator, TOSSIM, following which the Embedded Software group’s 24-node testbed was used. The crashed node and sensor value models described in Section 5.5 were used to model the system.

This chapter starts with an analysis of the memory used by the implementation in Section 6.1, followed by a description of the experimental setup in Section 6.2, and continues with experimental results. An analysis of an example test run is done in Section 6.3. Experiments were performed to measure the detection rate, including the number of false positives, the results of which are discussed in Section 6.4. Finally, the number of messages in the network containing conflicts were recorded, to show message overhead, which is discussed in Section 6.5.

6.1 Memory use

As the inference engine is meant to be used on top of an existing system, its memory use should be kept to a minimum. This section explores the amount of memory used by the engine in theory, and in practice.

6.1.1 Theoretical memory requirements

For the crashed nodes problem, three constraints are used, with the resulting memory use documented in Table 6.1.

The minimum space for the crashed nodes problem occurs if an observation is added that a beacon was received from the neighbouring node (an extra observation, $\text{receive}(\text{node}, \text{neighbour}) = \text{true}$ is added, superseding the previous prediction). The maximum memory use occurs if no beacon was received from the neighbouring node (an extra observation, $\text{receive}(\text{node}, \text{neighbour}) = \text{false}$ is added, but it does not supersede the previous prediction).

For sensor values, the maximum value occurs in a more complicated scenario. If a node receives a sensor value from one of its neighbours, without having recorded a

Table 6.1: Space needed, in bytes, for the inference engine databases. Note that the space for eight neighbours is not always a multiple of the total space needed for one node, as described in Section 6.1

	Observations			ID databases		Total (Max)		
	Model	Initial	Min	Max	Obs	Env	Model/Obs	+ IDDB
Crashed Nodes	15	18	15	21	20	13	36	69
Sensor values	5	0	0	17	6	6	22	34
Total	20	18	15	38	26	19	58	103
For 8 neighbours	160	144	120	323	187	131	483	801

Table 6.2: Space needed, in bytes, for the inference engine databases, without the use of the ID Databases

	Observations				Total (Max)
	Model	Initial	Min	Max	
Crashed Nodes	59	54	41	41	100
Sensor values	13	0	0	33	46
Total	72	54	41	74	146
For 8 neighbours	576	432	328	643	1219

sensor value of its own, it will predict its own value to be the same, with an environment size of two. It then proceeds to predict other neighbouring nodes' sensor values to be the same, with an environment size of three. For multiple neighbours reporting data, the size needed is further increased. For example, in the case of 8 neighbours, the memory thus used would be $i * ((obs_{size} + (obs_{size} + 2)) + (8 - i) * (obs_{size} + 3))$, with i equal to the number of observations from neighbouring nodes. The worst case scenario occurs at 5 neighbours, for 155 bytes, which is why the total for 8 nodes in Table 6.1 is not a multiple of 8.

The experiments done used a maximum of eight neighbours, for which the memory use is indicated in the bottom row of Table 6.1. Similar calculations were performed for the ID databases, the results of which can be found in the last two columns. Note that the total ID database size for eight neighbours is not a multiple of the previous total, since for the sensor value constraints, observation IDs for the local node are reused. Further, in an actual experiment, one extra byte is needed to mark the end of the databases.

Further calculations were also done on the actual savings due to the separate ID databases. The results of the same calculations performed for Table 6.1 performed without ID databases are presented in Table 6.2. Total memory use for the minimal case was 670 bytes with ID databases and 1024 bytes without them. In the maximum

Table 6.3: Size of the parameters used in the experiments (in bytes)

Neighbour Table	Model/Obs DB	Obs ID DB	Env ID DB	Symptom DB	Total
24	500	220	220	100	1064

Table 6.4: Image size as built with *make tnode*, space in bytes

	ROM	RAM	Extra RAM
No MBD	40890	1354	0
Leaf node	67366	2552	1198
Sink (serial enabled)	74584	3252	1898

case, 801 bytes were used with ID databases, and 1219 bytes without them. In both cases, the solution with separate IDs saves about a third in memory costs.

6.1.2 Actual memory use

The actual parameters used during experimentation are shown in Table 6.3. RAM and ROM use as reported by the TinyOS build system are shown in Table 6.4. The RAM use reported here is the memory used by state variables, and does not include any memory used on the stack. Although further optimisations could reduce the 1198 extra bytes reported for the proof of concept application, the size of the databases has already been heavily optimised. This means that in an actual deployment, the benefits of having a diagnostic system built in to a node must be carefully weighed against the memory requirements.

ROM use was further analysed, revealing that that the glue class used about 8K of ROM, in addition to an extra 10KB for the timers in the glue class. The inference engine used about 8KB. The large size of the glue class was due to the conflict sending code included there.

6.2 Experimental setup

Initially, a number of scenarios were set up for TOSSIM simulations. These included a seven by seven grid without noise, a densely populated scenario, a noisy environment, and four normal scenarios. The proof of concept application described in Section 5.6 was then run on these scenarios, and on the Embedded Software group's 24-node hardware testbed, PowerBench [5]. Ten runs were done for every scenario, and on the testbed.

The first scenario (referred to as grid) was an idealised seven by seven grid, where every node can reach only its direct physical neighbours (e.g. node two can reach nodes one, three, and nine). The second scenario (noisy) contained 49 nodes, randomly distributed on a fifty meter by fifty meter field, with a relatively high number of bad quality radio links. Every node was connected to 2.7 nodes, on average. The

third scenario (dense) tested a denser distribution with 49 nodes randomly distributed on a square with twenty-five meter sides. Nodes could hear, on average, 29 other nodes. Finally, four different scenarios (normal) were run, again with 49 nodes, on a fifty by fifty square meter field. These four scenarios were randomly generated, based on the same noise and distance parameters. These scenarios had nodes connected to an average of about 10 other nodes.

The model size, observation ID database size, and environment ID database size were kept constant throughout the tests. The amount of space reserved for these databases allowed constraints and observations to be stored for eight neighbouring nodes, using the example models described in Section 5.5.

For every run, the model was instantiated after 500 seconds. At fixed times, node crashes were triggered, and sensor faults were introduced. In total, three node crashes, and four sensor faults were introduced. To simulate a node crash, the radio was disabled when a fault was triggered. Sensor faults caused sensors that were initially producing normal values to switch to invalid values.

As discussed in Section 5.6, TinyOS's collection tree protocol implementation was used in the proof of concept application. CTP sends out beacons at varying intervals depending on the stability of the network, with a set maximum interval. To allow more runs to be performed in a smaller time frame, CTP's maximum beacon interval rate was decreased. In a real-life scenario, time spans would therefore be greater. A longer eight hour run was also executed, with standard CTP timing left intact. The results of this longer run can be seen in Figure 6.1, and are discussed in the next section.

6.3 Analysis of a single run

This section presents the results of a single longer run on the testbed. CTP's maximum beacon interval was left at its default level throughout this run, resulting in a maximum interval between beacons of 1024 seconds. The application waited four times this period for beacon messages from neighbours, or 4096 seconds, to reduce the number of false positives. This means that the maximum time to detect a crashed node should be 8192 seconds, or two intervals. This is due to the fact that if a node dies later in the first interval, it might have already sent out a beacon, marking it as alive during that period. As an example, node 11 died at 15000 seconds, but its death was only detected at 21700 seconds, 6700 seconds later.

A time-line of the events triggered during the run can be found in Table 6.5. Figure 6.1 shows the output of the eight hour run. The red and blue lines show the number of false positives against time, with very brief peaks for the sensors, which will be explained later in this section. No nodes were misdiagnosed as having crashed, so the number of node false positives remains at 0 throughout this experiment. The green and orange lines show the number of nodes and sensors which should have been found at a given time, but were not. The black line shows the total number of messages (including forwarded messages) sent up to that point in time.

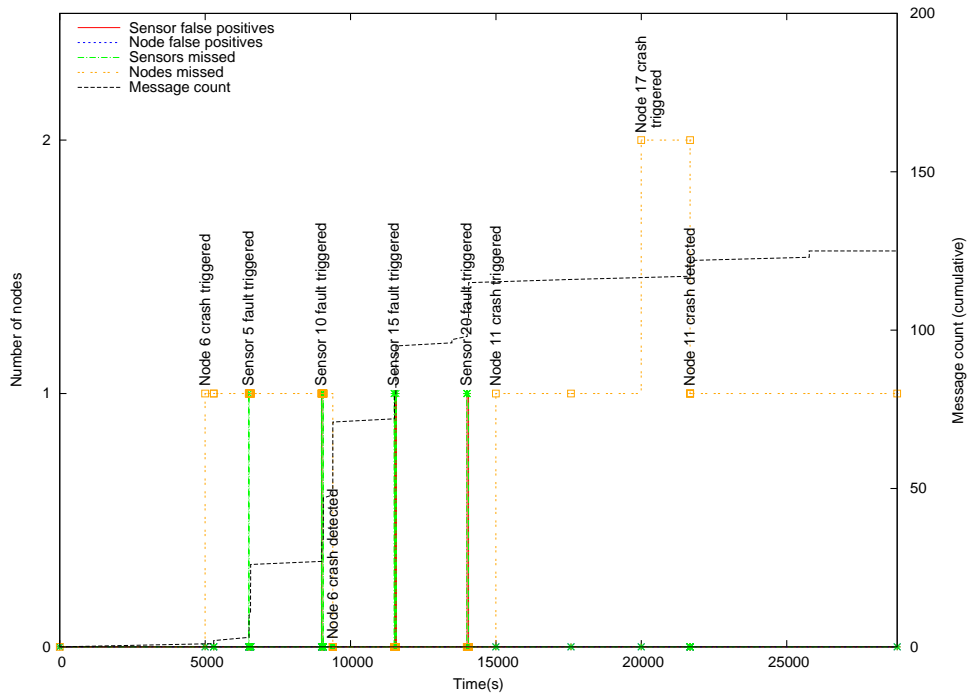


Figure 6.1: Time line of a longer run on the 24-node testbed.

Table 6.5: Time-line of triggered events during the 8-hour testbed run.

Time (s)	Event
1200	Model instantiated
5000	Node 6 crash triggered
6500	Sensor 5 failure triggered
9000	Sensor 10 failure triggered
11500	Sensor 15 failure triggered
14000	Sensor 20 failure triggered
15000	Node 11 crash triggered
20000	Node 17 crash triggered
28800	Run ends

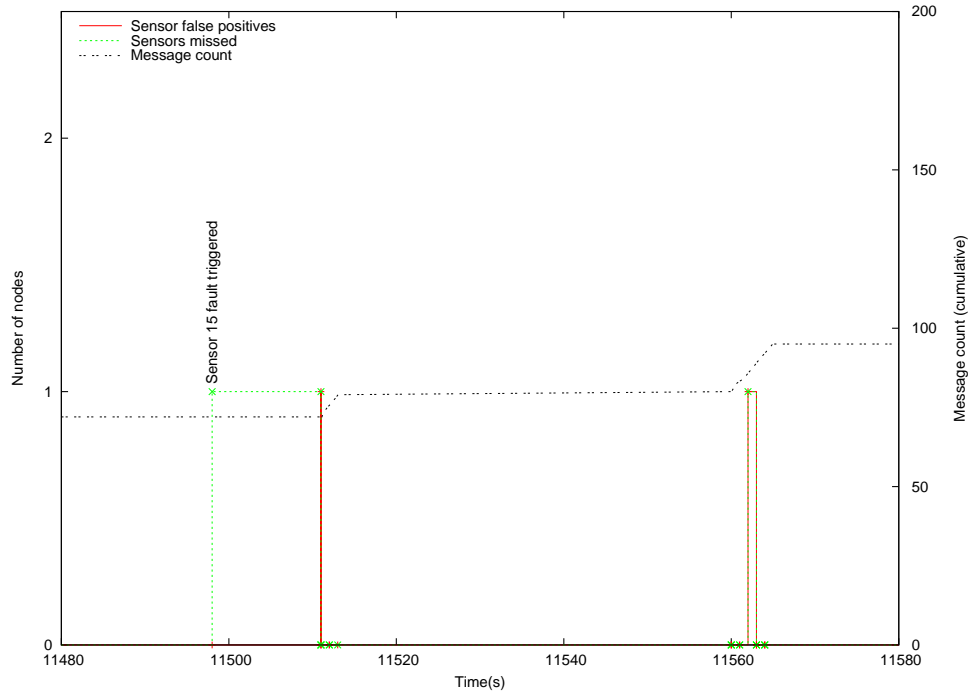


Figure 6.2: Detail of the detection of sensor 15's death, cf. Figure 6.1.

As can be seen, the failures of nodes 6 and node 11 were detected at 9400 and 21700 seconds, respectively. Node 17's failure was, however, never detected during this run. The failure to detect node 17 shows a shortcoming of the system. As every node stores constraints and observations for eight of its neighbours, it is possible that none of the nodes have a record for a certain node. This is what occurred for node 17. In an actual deployment, a more sophisticated method would therefore be advisable when instantiating the model, which will be further discussed in Section 6.6.

For sensor failures, details are not visible at these scales, so as an example of a sensor failure, take Figure 6.2, where sensor 15's death is highlighted. As can be seen, at around 11500 seconds, sensor 15 dies. About 10 seconds later, it sends out a packet containing its sensor values. Immediately after that, the inference engine on the node itself and its neighbours will detect a symptom, and a number of conflict messages are sent, as indicated by the black line in the graph. Initially, there is only one conflict between sensors 11 and 15, which means that a best diagnosis cannot be selected from the diagnoses [Sensor 11] and [Sensor 15]. Faced with two equal diagnoses, the diagnoser picked the first one, which very briefly resulted in a false positive. Soon after that, another conflict, <Sensor 15, Sensor 16> arrives, and the ambiguity is solved.

The second peak in Figure 6.2 results from the fact that a packet containing sensor values contains multiple values. The proof of concept application takes the average of these values to produce an observation. After a sensor switches to sending invalid

values, the first packet that is sent will often contain both correct and incorrect sensor values. The next packet will then contain only incorrect sensor values, resulting in a different average sensor value, and therefore a different value for the observation. Since this second observation overwrites the first, the initial conflict is invalidated, and a message is sent to the sink. The second observation then produces a new conflict, which is again sent to the sink. This situation is undesirable, as it results in unnecessary communication. Possible solutions are to force the node to delay sending conflicts until it has received a second set of sensor readings, or to only sample one observation from every packet received.

To further reduce the number of messages sent to the sink, the conflicts and conflict invalidations resulting from the same observation could be aggregated into a single message. Another advantage of doing this, is that the second peak in Figure 6.2 will disappear altogether, as conflict invalidations and the new conflict will always be sent together, since they result from the same observation. This would reduce the number of messages for the second peak by at least a factor two.

Although the brief sensor false positives discussed in this section were included in this section for demonstration purposes, they were later filtered out by having the sink-side diagnoser wait briefly for a second conflict before outputting its diagnoses.

6.4 Detection rate

To test whether the diagnoser produced reliable results, the results of the experiments discussed in Section 6.2 were analysed. The metrics stored were the number of false positives, the time it took to remove these false positives, the number of nodes or sensors missed by the diagnoser, and the time it took to find these nodes or sensors. The results are shown in Figures 6.3, 6.4, and 6.5.

Sensor false positives are shown in Figure 6.3. The undetected false positives in the noisy runs are due to conflict messages not reaching the sink. The false positives in the dense scenario had a different source: due to the density of the network, most of the links in CTP's neighbour table were not stable yet at model instantiation time. Model instantiation skipped these links, causing very few constraints to actually exist in the system. Only one node reported a conflict for one of the sensors, and the diagnoser picked the wrong entry out of that conflict as the best diagnosis. The outliers between 40 and 60 seconds are nodes that were detected in a second round, when a neighbouring node did not overhear a sensor packet in the first round.

Only very few node false positives occurred, and as such, no graph of them is included. A node false positive did occur once during normal scenario three, for 400 seconds (the interval between two beacon checks). It was due to two links to a node failing simultaneously, which caused the diagnoser to think that the node was at fault. The false positive could have been prevented by changing the way in which the best diagnosis was selected. However, that might cause more nodes to be missed in other scenarios. Three further false positives were recorded for the noisy scenario, again because of noisy communication links.

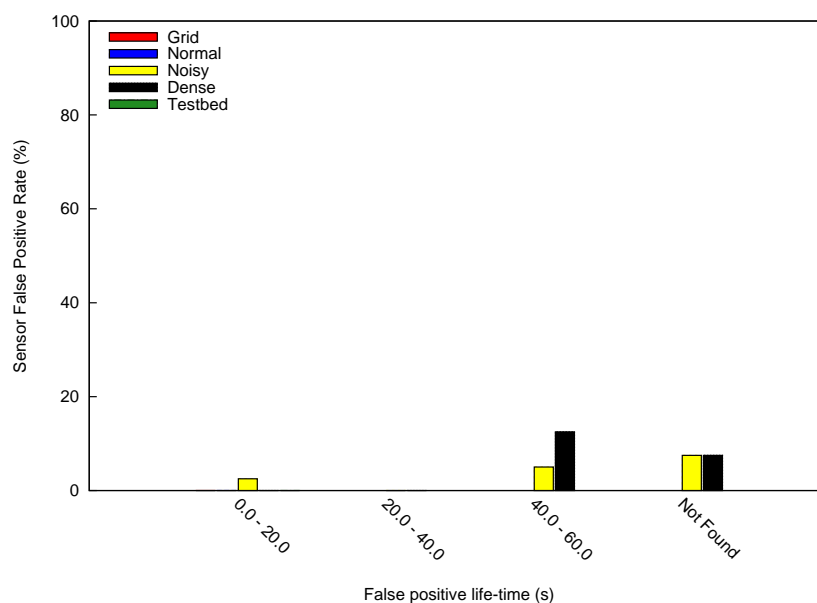


Figure 6.3: Sensor false positive rate per scenario, split by lifetime of the false positive.

The time taken to detect node failures, as shown in Figure 6.4, generally agrees with the expected outcome. The peaks for node failure are between 400 and 700 seconds, which is in the second period of beacon checks after a node fails, as explained in Section 6.3. The large number of missed failures in the noisy and dense scenarios occur for the same reasons as the sensor false positives. The failure to find nodes in the normal scenarios and the testbed was due to the nodes being poorly connected to the network, which caused neighbouring nodes to skip them during model instantiation.

Figure 6.5 shows sensor detection. The delay before a sensor was detected is due to the fact that time was measured from the moment a failure was triggered. As there is a delay before sensor measurements are actually sent, there is a time gap between sensor failure and detection. In general the results are as expected, with the noisy and dense scenarios underperforming again.

Another issue came up while performing these tests: in the case of heavy noise, and without checking link quality when instantiating the model, a large number of conflicts are sent due to bad links. This caused the sink-side diagnoser to slow down a great deal, up to the point that analysing data became impossible. This is due to the brute force approach used in solving the minimal hitting set problem. In an actual deployment, an algorithm that calculates an approximation of the minimal hitting set is therefore advisable.

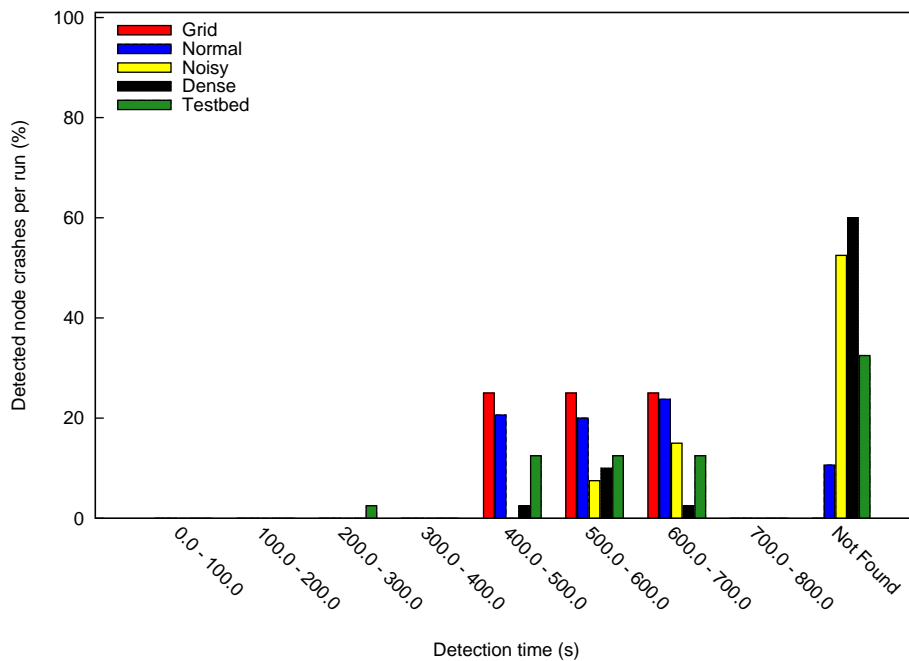


Figure 6.4: Time taken to detect node failures.

6.5 Communications Overhead

The number of messages passed is a metric closely related to power consumption, as radio transmissions are the most costly operation on a sensor node. Figure 6.6 shows the average number of messages sent and forwarded in the network for each of the scenarios. As can be seen, the noisy and dense message count is lower, due to their smaller model. The simulated runs produce a lot more messages than the testbed runs, due to the fact that nodes in the testbed are almost always within one or two hops of the sink.

The number of messages sent is currently relatively high, with an average of 70-90 messages required for a single failure detection in the normal and grid scenarios. As can be seen in Figure 6.1, messages are only sent when a symptom is detected, and only happen when new observations come in. The heaviest message load occurs on sensor detection. This suggests that too many nodes are checking their neighbour's status. A further source of messages is the two different observations coming in for a single change of sensor value, due to the averaging explained in Section 6.3. An argument can therefore be made for a delay before sending conflicts to the sink.

6.6 Discussion

Under normal circumstances, the system performed well. However, a few problems did crop up during the dense and noisy runs, mostly due to the model, and the way it

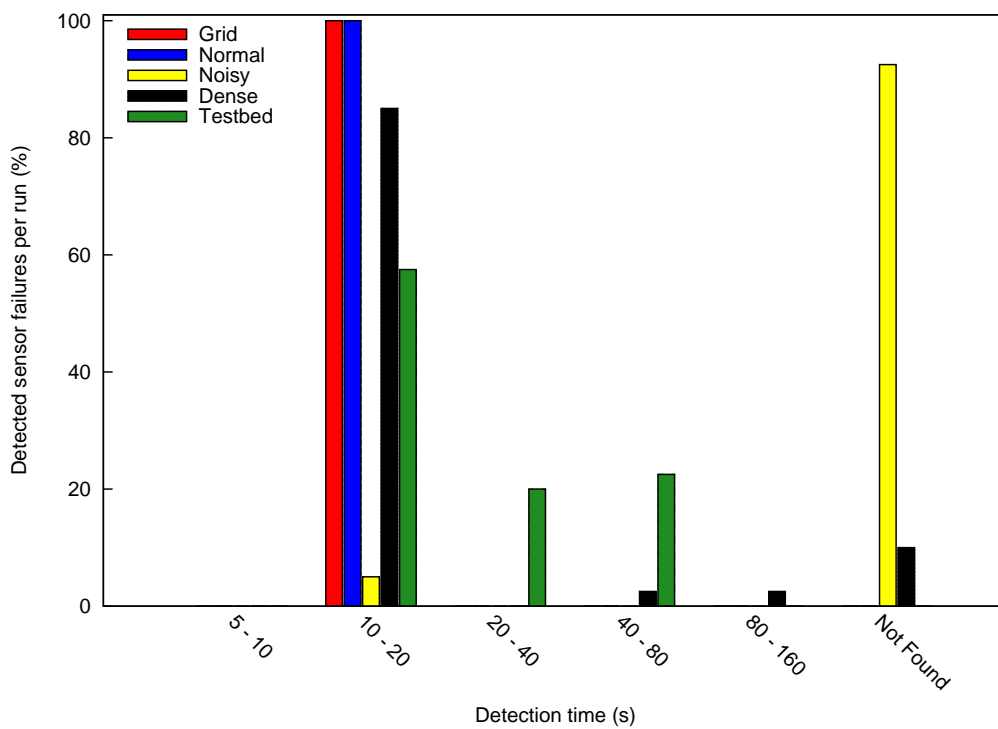


Figure 6.5: Time taken to detect sensor failures.

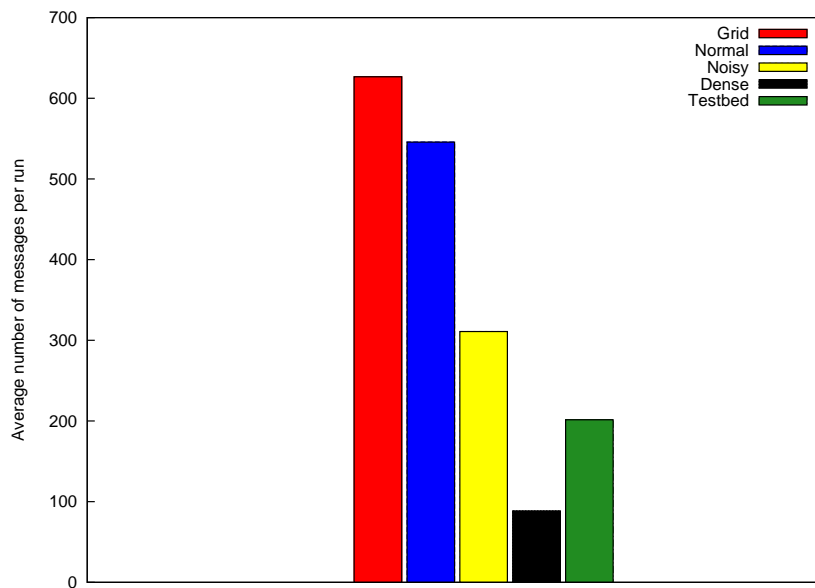


Figure 6.6: Average number of messages sent and forwarded for every scenario

was instantiated.

Model instantiation worked well under normal circumstances, such as on the testbed, and during the normal scenarios. Model instantiation still requires a serious look, however, as the dense network scenario's failure showed. Due to the fact that the nodes' neighbour lists were still very much in flux, very few nodes were added during model creation. At the very least, the time at which a static model is fixed in place needs to be carefully determined.

Perhaps a more dynamic model could be used, that continually adapts its constraints to the circumstances outside of the network. For example, one could imagine that the constraints in the model were regularly updated, depending on the status of a node in the neighbour list. Care would however have to be taken that, for example, crashed node detection did not cease to work because a node was removed from the neighbour list, and therefore from the model.

The number of messages needed for diagnosis was reduced to only conflict messages. However, further research needs to be done to further reduce the communications overhead. Again, model creation is partially to blame, since sometimes a large number of neighbouring nodes are watching each other. On the other hand, in other cases, nodes were not watched by any neighbours at all. A potential solution to this problem would be coordination between neighbouring nodes during model creation, using some form of topology control. Extra coordination would require extra communications, however, meaning that a good balance must be made for different constraint types. A general survey on topology control can be found in Santi et al. [15].

Another source of extra messages are false positives. These could be prevented by having nodes wait for a set period of time before sending out conflicts, allowing newer conflicts to supersede them, or conflict invalidations to be generated. Further message reduction can be achieved by using aggregation, as one observation can cause multiple conflicts and conflict invalidations to be generated.

Finally, a more general observation that was made during testing was that diagnosis calculation at the sink worked well under normal circumstances. However, when a high number of radio links failed, too many conflicts arrived at the sink, and the amount of time taken to calculate diagnoses became too high. This was due to the brute force approach to calculating the minimal hitting set of the diagnoses. Replacing this algorithm by an approximating algorithm would solve this problem.

Conclusions and future work

The primary goal of this thesis was to investigate the viability of a model-based diagnoser on wireless sensor networks. As model-based diagnosis is a general diagnosis technique, it can detect multiple faults of different types. This allows a single diagnoser to be used where previously multiple ad-hoc solutions would be used.

A design for such a model-based diagnoser was presented, based on the idea of splitting the diagnosis process into two parts, similar to de Kleer's general diagnostic engine (GDE)[3]. The first part of the diagnoser consists of an inference engine, which was designed to work within the memory and CPU restrictions of sensor nodes. The inference engine runs on every node in the network, using a simple local model and locally gathered observations to produce conflicts, which are sent to the central sink node. The second part of the diagnoser runs on a more powerful computer connected to the sink node, and computes the minimal hitting sets of the conflicts to produce diagnoses.

This diagnoser was then exercised using a proof of concept application. This application allowed sensors to be triggered to produce invalid values, and node crashes to be simulated. A number of experiments were performed on the application, to measure detection rate, communications overhead, and memory use. The results in the previous chapter show that the design described in Chapter 4 is a valid model for a diagnoser in most cases. Given a few correct observations, the inference engine generated the right minimal diagnosis. A number of problems were encountered in dense and noisy environments, which were due to the way the model was instantiated and packet loss in the network.

Communications overhead was still relatively high, with about 70 to 90 messages being either sent or forwarded per introduced failure. This can, however, be reduced by waiting before sending a conflict to the sink, to see whether the conflict remains valid over a longer period of time. Further, aggregating multiple conflict messages on the node can also reduce the number of messages that need to be sent.

Memory use was reduced as much as possible for the inference engine's databases, ensuring that enough memory was available for the normal application of sending sensor information back to the sink. In more complex applications, however, the

memory overhead might prove to be too high.

In conclusion, using model-based diagnosis in sensor networks is certainly a viable solution, and has the potential of greatly simplifying the diagnosis process. The system is flexible enough to support different constraint types, allowing a wide range of applications.

7.1 Future Work

A number of additions to the current diagnoser could be introduced in future work. Currently, once a set of diagnoses has been calculated at the sink, it is reported to the user, and no further action is undertaken. A future direction of research could be to actively send queries into the network, to confirm or reject some of the generated diagnoses. Along a similar vein, observations could be exchanged between different nodes, to increase the chance that a problem is found.

Automatic generation of observations, models and environments based on simple constraints would further increase the ease with which the diagnoser can be used in future deployments. Currently a small number of data structures needs to be modified to add new observations or environments. For new constraints, in addition to the modification of a few data structures, a small piece of code needs to be added that makes predictions for the inference engine. Although this is not a difficult process, the generation of these data structures should be relatively easy to automate. Automatically generating the code could prove more difficult.

Another interesting direction of research would be into more dynamic models. Changing the model at run-time might prevent some of the problems that occurred during model instantiation, such as the lack of a stable list of neighbours in the dense scenario used during experimentation. Care should be taken not to throw away too much information during dynamic model generation, however. For example, if a node is no longer in the neighbour list due to a lack of beacons from it, throwing away the constraints for that neighbour would cause a failure to detect its crashed status.

Finally, further message reduction might be achieved by combining conflicts into partial diagnoses in-network. This would allow aggregation of conflicts, not only reducing the message count, but also reducing the load on the sink node. It is however questionable whether such aggregation would not be too costly in both memory and processor use.

Bibliography

- [1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.
- [2] J.I. Choi, J.W. Lee, M. Wachs, and P. Levis. Opening the sensor network black box. *ACM SIGBED Review*, 4:13–18, 2007.
- [3] J. de Kleer and B.C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, Apr. 1987.
- [4] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 1–11, June 2003.
- [5] I. Haratcherev, G. Halkes, T. Parker, O. Visser, and K. Langendoen. PowerBench: A scalable testbed infrastructure for benchmarking power consumption. In *Int. Workshop on Sensor Network Engineering (IWSNE)*, pages 37–44, June 2008.
- [6] M. Kalech and A.G. Kaminka. Towards model-based diagnosis of coordination failures. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 102–107, 2005.
- [7] J. Kurien and X. Koutsoukos. Distributed diagnosis of networked embedded systems. In *Proceedings of the 13th International Workshop on Principles of Diagnosis (DX-2002)*, pages 179–188, 2002.
- [8] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *14th Int. Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*, April 2006.
- [9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.
- [10] K. Lorincz, M. Welsh, O. Marcillo, J. Johnson, M. Ruiz, and J. Lees. Deploying a wireless sensor network on an active volcano. In *IEEE Internet Computing*, volume 10, pages 18–25, 2006.
- [11] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, 2002.
- [12] L. Paradis and Q. Han. A survey of fault management in wireless sensor networks. *Journal of Network and Systems Management*, 15:171–190, June 2007.
- [13] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 255–267, 2005.
- [14] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, Apr. 1987.

- [15] P. Santi. Topology control in wireless ad hoc and sensor networks. *ACM Computing Surveys*, 37(2):164–194, 2005.
- [16] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks*, pages 121–132, 2005.
- [17] P. Zoetewij, J. Pietersma, R. Abreu, A. Feldman, and A.J.C. van Gemund. Automated fault diagnosis in embedded systems. In *Proceedings of the 2008 Second International Conference on Secure System Integration and Reliability Improvement*, pages 103–110, 2008.