Passive distributed assertions in wireless sensor networks

Passive Distributed Assertions in Wireless Sensor Networks

Master Thesis

Philippe Jacot
<jacotgup@student.ethz.ch>

Supervisor: Kay Römer

Prof. Friedemann Mattern
Distributed Systems Group
Institute for Pervasive Computing
Department of Computer Science

September 10, 2008

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

After deploying a wireless sensor network (WSN) developers often find that their application does not meet the requirements, even though it worked flawlessly when deployed on a testbed. This gap between the laboratory and real-world settings is caused by the changed environmental influences that might improve the communication between two nodes or prevent it altogether. Software engineers often find it hard to trace an observed malfunction back to the underlying causes. Techniques usually applied in other fields to track down errors in computer software fail in the context of WSN. Due to resource constraints, it is hard to inspect the internal state of a single node of the network and errors are often caused by multiple nodes, whose collective state leads to the encountered malfunction.
This thesis proposes *PES*, a system that aims to make debugging of wireless sensor networks easier by monitoring the application and checking its health using developer-provided distributed assertions. *PES* needs no wired connection to the nodes in the network and is therefore applicable for testing in a lab as well as for inspecting a WSN in the field.

Zusammenfassung[1]

Nach der Ausbringung von drahtlosen Sensornetzwerken (DSN) stellen Entwickler häufig fest, dass eine Anwendung, welche noch in der Testumgebung tadellos funktioniert hat, nicht mehr zufriedenstellend arbeitet. Diese Diskrepanz zwischen Labor und Realität ist auf die unterschiedlichen äusseren Einflüsse zurückzuführen, welche die Kommunikation zwischen einzelnen Knoten mal verbessern, sie jedoch in anderen Fällen komplett verhindern. Den Entwicklern fällt es oft schwer ein beobachtetes Fehlverhalten auf die zu Grunde liegenden Ursachen zurückzuführen. Techniken, welche von Ingeneuren in anderen Gebieten der Software-Entwicklung üblicherweise verwendet werden, versagen im Zusammenhang mit DSN. Dies weil sich der Zustand einzelner Sensorknoten wegen derer beschränkten Ressourcen nur mit Mühe auslesen lässt und Fehler häufig durch Wechselwirkung mehrerer Knoten verursacht werden.
Das in dieser Masterarbeit vorgestellte System *PES* verfolgt das Ziel, die Fehlersuche in drahtlosen Sensornetzwerken zu erleichtern, indem es die Ausführung der Anwendung überwacht und verteilte Zusicherungen (*engl. assertions*) überprüft, um fehlerhafte Netzwerkzustände zu erkennen. Das Ganze geschieht ohne Verkabelung der Knoten, damit *PES* auch nach der Ausbringung des Netzes eingesetzt werden kann um Funktionsstörungen zu untersuchen oder die Ausführung der Applikation zu überwachen.

[1] ETH regulations require the abstract to be given in a national language.

Acknowledgments

I'd like to thank Prof. Friedemann Mattern, his research group for giving me the opportunity to write my master thesis in the field of distributed computing. Special thanks go to Kay Römer, who supervised this thesis and provided my with many useful ideas for the design and valuable feedback on the this documentation.

Affidavit

I hereby declare that this master thesis has been written only by the undersigned and without any assistance from third parties. Furthermore, I confirm that no sources have been used in the preparation of this thesis other than those indicated in the thesis itself.

This thesis has not yet been presented to any examination authority, neither in this form nor in a modified version.

Place, Date

Signature

Contents

1 Overview

A wireless sensor network is composed of a large number of autonomous devices, also called nodes, that communicate among each other using a radio transceiver. The nodes in such a network are typically limited in both energy and computing power, and thus have to be programmed and configured in a way that enables them to carry out their aim in an efficient manner. To allow nodes to monitor their surroundings, they are equipped with sensors that enable them to measure a wide variety of environment parameters and thus gather the necessary information to fulfill their function. Even though nodes are usually deployed for a single purpose, their hardware is designed to allow their utilization in various scenarios and their software needs to be adapted for every application to yield the best possible results.

1.1 Problem Description

The challenges involved in writing software for a wireless sensor network (WNS) are twofold. A node in a sensor network is an embedded system, without a large memory or a fast processor. These limited resources restrict an application programmer when writing the application as well as during the process of debugging. The limited bandwidth that is available to connect the node to a PC makes it difficult to interactively debug the code as it is accustomed to applications targeting a more potent architecture. It is also not possible to record the execution and later transfer the application trace to a computer for offline analysis since there is not enough memory for a long trace. Additionally, the insertion of code that facilitates debugging, always comes at the risk of changing the runtime behavior of the application and therefore introducing new or hiding present bugs. Although debug code might introduce bugs on any system, this is particularly the case when a considerable part of the CPU time is spent executing debug code, which more easily is the case with slow processors.

The other difficult aspect is the distributed nature of a WSN. To keep track of the state of the application, it is not sufficient to monitor a single node, as it is only a part of the network and therefore only holds part of the information on the state of the application. An external observer wanting to keep track of the application needs to monitor all nodes and merge the knowledge about the single nodes into a global picture. The distribution also entails various disadvantages to the single nodes. To be able to perform the collective goal, the nodes need to interact and coordinate using wireless communication. But this communication is not reliable as external factors significantly influence the propagation of radio signals and thus can render communication between two nodes impossible.

When a newly developed application proves to work satisfactorily in a lab testbed,

it is deployed in the target environment. But the change of environment often introduces new problems that lead to reduced performance or complete failure of the sensor network. If such a situation arises, a developer needs some clues to identify the cause of the observed behavior, in order to be able to fix the problem or at least reduce its impact. Unfortunately, gathering information about the network is even tougher in the field, as the nodes run on battery and cannot be easily attached to a monitoring device.

1.2 PDA Evaluation System

The term PDA stands for passive distributed assertion as defined in [8]. The *distributed* should express the ability of such an assertion to be formulated over the distributed state of a wireless sensor network in which all nodes hold a different part of the application state. The *passive* refers to the underlying technique used which passively inspects a sensor network by capturing the messages coming from the nodes. The PDA evaluation system, or *PES* for short, developed in the scope of this thesis should help to detect erroneous behavior of a sensor network and enable a developer to identify the underlying cause. To this end, *PES* allows a developer to insert the above mentioned PDAs into the source code of the WSN application, to express the assumptions made about the whole network's state. The inserted assertions are broadcast by the node, together with information about the node's current state. Dedicated sniffer nodes collect these messages and forward them over a secondary, independent network to a central computer that evaluates the assertions using the state of the network, also collected using these sniffer nodes.
When evaluating an assertion, *PES* considers inherent difficulties in distributed system, such as the lack of perfect time synchronization and the possibility of incomplete network state information. This thesis aims to make the following contributions:

- A language, designed to be used in distributed assertions.
- An algorithm for verifying distributed assertions which takes missing state information and synchronization errors into account.
- Design, implementation and evaluation of a system that incorporates the above mentioned contributions, including a runtime system for BTnodes and a backend for the evaluation of distributed assertions.

1.3 Structure

In chapter 2 of this document, different solutions addressing the problems outlined in this chapter are presented and compared to the solution provided by this thesis. Chapter 3 describes the design of *PES* and how it addresses the inherent challenges of wireless sensor networks, whereas chapter 4 focuses on the corresponding implementation. An example application that was developed to test and evaluate *PES* is described in chapter 5 and the results of the conducted evaluation can be found in chapter 6. Chapter 7 closes this thesis with a review of the achieved goals, fundamental limitations and suggestions for further improvements.

2 Related Work

This chapter discusses work related to the idea of passive distributed assertions for sensor networks. The first part focuses on other ways of debugging sensor networks, while the second part gives further insight into assertions used in conjunction with distributed systems.

2.1 Debugging Sensor Setworks

There are several approaches to facilitate the debugging of wireless sensor networks (WSN) and to allow faster detection of errors and their underlying causes. In this section other techniques are described that were designed to ease the development and deployment of sensor networks.

2.1.1 Wringer

In [9], published while this thesis was being written, Tavakoli et al. propose a system that introduces predicates into the nodes of a sensor network, which serve to identify node states of interest. The predicates list the conditions that have to hold in order to execute an associated action, for example, sending a message to a network observer. Wringer makes the variables of a node directly accessible by extracting all symbols of program variables and their location in the node memory from the application source code. These symbols can then be used to refer to a variable on the local node in a predicate. They also mention the possibility of accessing variables on other nodes found in the network. The developer has only to specify which variables have to be monitored and Wringer takes care of inserting breakpoints whenever a new value is assigned to the variable. The breakpoint then triggers the evaluation of the assigned predicate and, if the predicate's conditions are met, executes the associated action.

Wringer allows predicates to be added to a node during runtime by sending it a special message that installs the predicate and the accompanying breakpoints, called watchpoints, onto the node. This frees the developer from recompiling the application and reprogramming a node with it, just because he wants a predicate changed or devised a new predicate to narrow down an error in the wireless sensor network.
On the other hand, assertions, as proposed by this thesis, are embedded in the source code and have to be compiled with the application. Therefore, the introduction of a new assertion requires reprogramming the nodes. The inclusion of assertions into the application's source code also has advantages, such as the comfort that the programmer can specify the assertions alongside with the application code. He does not have to worry about the possibility that the assertion is evaluated under unforeseen conditions, as the context that the assertion is evaluated in, is clearly defined by its

location in the source code. Additionally it allows the use of values in the assertion that are not stored as global variables and are, for example, only valid for the time of a function invocation.

Another difference concerns the communication from Wringer/*PES* to a central computer. While Wringer uses the resources of the sensor network itself, it imposes much more interference with the actual application compared to *PES* which only broadcasts the messages used for collecting state information and assertion evaluation, which are then forwarded using a secondary, independent deployment support network.
Wringer and *PES* also differ in the location where an assertion or a predicate is evaluated. Wringer is essentially a basic Scheme interpreter that allows the evaluation of the predicates directly on the node and only communicates to collect variable values from other nodes and to notify a satisfied predicate. *PES* on the other hand only triggers the evaluation of assertions on the nodes and a central computer, which also collects the states of the nodes and performs the evaluation without using node resources.

2.1.2 MEGS

The application MEGS [6], that, like Wringer, was introduced on this year's HotEmNets in June, collects the state of the nodes in a similar fashion to *PES* and stores the nodes' changing variables on a central computer. In both systems, the nodes indicate when they reach a location in the source code that has an assertion associated with it which should be evaluated using the node states that have already been collected. In contrast with *PES*, MEGS allows an assertion to be violated for a short time, defined by the user, without announcing an error.

A difference can be found in the way *PES* and MEGS represent the state of the network. While *PES* manages the state of the nodes on its own, solely based on the available attributes, a developer using a preliminary implementation of MEGS needs to manually create the node representation containing the available attributes by subclassing the general node class. The two applications also differ in the representation of the assertions. Assertions in MEGS are directly handled by a developer-provided class, which, based on the ID of an assertion, has to check whether the assertion is valid for a node or not. *PES* on the other hand extracts the assertions that are given as a Boolean expression in text form directly from the source code and then evaluates the extracted assertion whenever a node comes across the location of the node.

As MEGS follows the same approach to identify anomalies in a WSN application, they also encounter the same difficulties, such as an incomplete network state and an inaccurate clock synchronization. *PES* already takes these difficulties into account, while no such capabilities were mentioned for MEGS and it thus might simply return a wrong evaluation result.

2.1.3 Marionette

With Marionette [10], Whitehouse et al. propose a system that allows direct access to a deployed node with RPC-like semantics. It not only allows the variables to be read and executes methods on the node - or the nodes, if all nodes are accessed si-

multaneously -, but also makes it possible to directly write to a node variable. This enables an application developer to try different settings when running his application without having to change the node code or upload the new software. The direct modification of variables, however, requires deep knowledge of the application to prevent severe consequences. The real application might only change a variable in accordance with other preconditions or, alternatively, the application would usually acquire a lock when changing a variable to ensure a consistent node state.

While Marionette is convenient for inspecting nodes and their state, it does not ease the monitoring of the whole network and the overall network state. When various variables have been read from the nodes, these values still need to be interpreted. Additionally, the values retrieved are not synchronized, that is, the values are the results received after flooding the network with the request, with some nodes responding earlier than others. Marionette also imposes additional messages to the network as even requests to a single node are sent by Drip, an epidemic protocol that floods the whole network and therefore might hinder the real application's execution. The same doubts might be applied to *PES* as well, but in contrast with Marionette, only the originating node broadcasts a *PES* message. The packet is then captured and transferred to the collecting computer using an independent network built by dedicated nodes. As every communication with a node is initiated by the central workstation, a variable that should be monitored for changes needs to be polled at regular intervals, thus introducing overhead to the network, with the possibility of missing certain changes when they occur between two polls.

2.2 Distributed Assertions

The detection of global predicates has received much attention in the field of distributed systems. An assertion is basically a predicate which is restricted to values of a given point in time and the attached meaning that, under correct execution of the program, the predicate always holds. To prove that a predicate always holds, or formulated differently, that the negated predicate never evaluates to true, is not efficiently possible for predicates in general, as shown in [3]. A less ambitious goal is to validate that for a running or a previously ran computation, the assertion never was violated, that is to say, that the examined computation did not show an error. This goal is pursued by *PES*.
In distributed computations, the lack of synchronized clocks is usually compensated by logical time, in which, instead of the absolute time, a time is available that correctly represents the causality, that is the correct ordering of cause and effect. Since in a sensor network the cause of an action of the system is usually external to the system, environmental change for instance, logical time is of limited use, as the external event might be detected by multiple, autonomous processes, which cannot reliably assign all their activity to the single cause. To get a global state that is as consistent as possible, *PES* tries to map all states of the participating processes to a global clock. When evaluating an assertion over this global network state, there are three possible results: Besides the two obvious ones *fulfilled* and *violated*, there is also *unknown*, which indicates that the state of a node used in the assertion could not reliably be

detected.

There are a number of different approaches to evaluate assertions during a computation in a distributed system. One specific approach in the context of robotics is given below.

2.2.1 Distributed Watchpoints

In [4] De Rosa et al. propose a variation of assertions to debug multi-robot systems, called distributed watchpoints. Such watchpoints specify the error conditions that should be detected and are limited to conditions that can be expressed using a fixed-size, connected sub-ensemble of robots in order to prevent the exponential search that would be needed for disconnected sets. The watchpoints can then be specified using temporal constructs to refer to a node's previous state, spatial constructs to include the neighborhood of nodes and, finally, numerical state variables of the nodes.
With the ability to express watchpoints over the changing robot state, they provide a powerful tool for specifying possible error conditions. The limitation, however, that an error condition must be formulated using a fixed number of robots, which, additionally, have to be directly connected, might render the checking of certain error conditions infeasible. These restrictions might be acceptable for multi-robot systems, where predominantly neighboring nodes communicate and achieve a goal together. In a large scale sensor network, where interaction can take place between two arbitrary nodes, the limited node coverage for error conditions would presumably miss certain error patterns.

3 Design

The approach taken in *PES* to improve the understanding on what is happening in a sensor network, and how well it conforms to the expectations of the application developers is largely based on [8]. To enable an external observer to gain knowledge of the state of the network, every node publishes part of its internal state. Whenever a decision on a node is taken that is based on the (assumed) state of some other - not directly accessible – node, this can be stated in the source code such that a central instance that also collects the node state information is able to verify these conditions and might inform a developer about requirements that are not matched. The communication from nodes to the central point is unidirectional such that the nodes publish their state but neither know if anyone is listening, nor ensure that the state is delivered in a reliable fashion. This chapter will further elaborate on concepts and software design decision for the two part node integration and assertion evaluation. Chapter 4 will focus on implementation specific issues and will discuss various technical aspects of the two software components.

3.1 Overview

The state publishing is done using snapshots, while the conditions are specified in the form of assertions which are both directly embedded in the application code. As the assertions are not intended to be verified on a single node and the state is distributed among the nodes these assertions are called passive distributed assertions or PDAs for short.

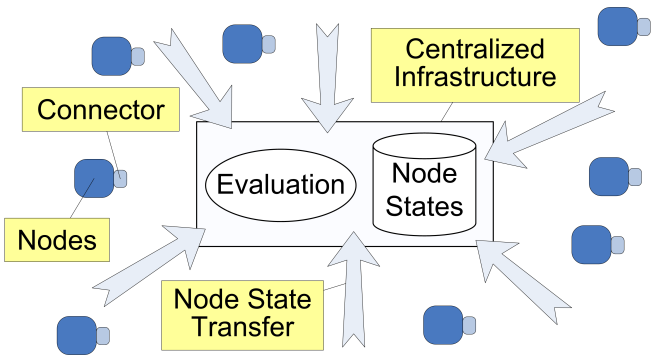

Figure 3.1: All nodes submit some of their state to a central instance that then evaluates the network condition.

To allow the gathering of data with minimal impact on the application code, a connector is introduced which offers the two main primitives for taking snapshots and stating

assertions. This connector then handles the transferring of the necessary information to the evaluating instance. In order to get a consistent picture of the network state, it is necessary to have a global clock. This allows the snapshots that carry changes of the nodes' states to be placed in relation to each other and makes it possible to evaluate assertions at a fixed point in time.
On the other hand there needs to be a central data collection point that stores the snapshot information and makes it available to be used in assertion evaluation. Figure 3.1 shows the basic architecture of the PDA evaluation system or *PES* for short. Figure 3.2 outlines how the central evaluator aligns the snapshots and assertion with a global clock.

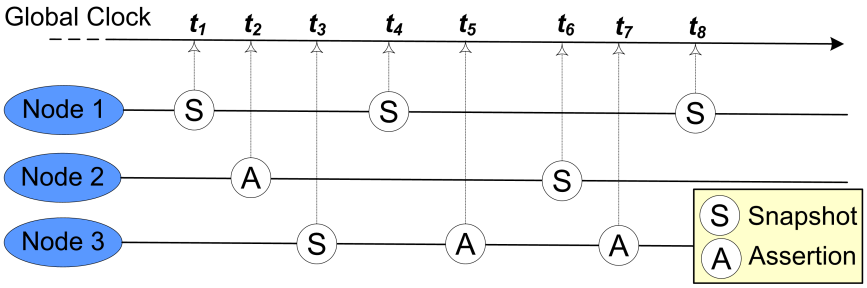

Figure 3.2: The central instance merges the snapshots and assertions of different nodes on a single timeline.

The transfer of state information and assertions is done using radio communication and SNIF ([7]) is used as a solution for overhearing and forwarding the data to a central point. To achieve this, the various SNIF nodes build up a *deployment support network* (DSN), which is then used to forward the message in the tree structured DSN until it reaches the root and thus the central instance. The SNIF nodes use their Chipcon module to collect node communication and Bluetooth to communicate among each other and with the application that collects and evaluates the incoming messages. Another important service provided by SNIF is its ability to assign timestamps to messages such that they can be chronologically ordered thus the needed time synchronization is provided by SNIF as well.

3.2 Concepts

This section introduces and describes the concepts used in the context of this thesis and the accompanying documentation.

3.2.1 Node

A node is in a broader sense a part of the network and interacts with other nodes within the network to achieve one or multiple common goals. Nodes have a state directly accessible only to them which is available in the form of variables that are situated somewhere in the nodes' memory, the precise location of which is no concern to an external observer. The developer can chose the part of the node state that is published and thus made accessible in a read-only fashion to an observer in the form

of a set of attributes that map a symbolic name to a value. It is not necessary that the nodes have the same software running on them, but it is required that each node has the same attributes that carry the same semantic meaning. Every node is assumed to have a unique ID in the form of an Integer that distinguishes it from the other nodes.

3.2.2 Attribute

An attribute is part of the node state that can be referenced using a symbolic name. Every attribute has a type that defines the context in which the attribute can be used within assertions and which values the attribute can take. A numeric attribute, for example, can be used with arithmetic operators while a Boolean attribute can be used in conjunction with Boolean operators.

3.2.3 Node Neighborhood

When talking of the node neighborhood, the network neighbors of a node are implied. These are the nodes that a given node can reach using its radio communication module. The node neighborhood - or simply *hood* - is not required to be static. It may change over time as nodes appear and disappear in the network or as external conditions influence the range of the node's Chipcon radio module. Although basically an ordinary attribute, the *hood* is treated in a special way as it is an inherent property of a node in a sensor network and the only attribute of the type *set*, that is an unordered enumeration of node IDs.

3.2.4 Snapshots

A snapshot assigns values to one or multiple attributes of a given node. It contains a list of string value pairs where the string is an attribute name and the value of a type valid for the attribute's type. The meaning of a snapshot is that *now* on the *current node* the listed attributes have the supplied values. This *current state* is published in the form of a message broadcasted via the Chipcon module such that it can be collected by listening to the messages sent by nodes.
Snapshots do not have to be taken manually. Assuming that an attribute is connected to a variable in the application program and no aliasing is allowed, it would be viable to take a snapshot whenever the variable value changes. To do so, it is sufficient to locate every assignment to the variable and insert a snapshot after the assignment that takes the new value for the variable and publishes it. This, however, is not accomplished in the scope of this thesis.
The snapshots described in this section are intended to allow an external observer to collect the state of the nodes by listening to their snapshot messages and, in this way, detect all nodes that are running in the sensor network as well as the values of their attributes. As the listener and the nodes run independently, the listener might be started after the network and thus has to catch up with the nodes' states.
When a central assertion evaluator starts listening to a network it takes some time until it has captured a snapshot from every node in the network, let alone collected the whole state of every node. Until every node and its state is known it is not always possible to evaluate assertions, especially when they involve a considerable part of the

node set, like for example an assertion to check that only one leader exists.

In order to have some upper bound by which time each node's state is collected, periodic updates are introduced. These updates behave like an ordinary snapshot containing the values of all node attributes. The difference from snapshots as described above is that the values given for the attributes are cached values from previous, developer invoked snapshots. This then makes it possible for periodic updates to be broadcasted whenever needed and allows periodic updates to be sent at regular intervals. These updates have two other benefits: firstly they compensate for lost messages such that after some time, even if a snapshot gets lost, the listening system receives the correct value. Secondly, they provide a heartbeat for nodes that allows any listener to detect defective or discharged nodes, since in contrast to the absence of ordinary messages, which are not necessarily sent in regular intervals, missing periodic updates indicate a defunct node.

Besides the normal attributes, the node neighborhood, as described above, is a special attribute that is handled in a specific way. The *hood* can change like every other attribute and thus there has to be a way to include it in a snapshot. For this reason the keyword *hoodset* is added for snapshots such that a developer who wants to make a snapshot of a changed hood can just include this keyword in the list of attributes for the snapshot and the snapshot will contain the current set of neighboring nodes. No additional value has to be provided for the *hoodset* attribute, the neighborhood is gathered via a function which is registered by the node application developer during initialization of the sensor node. This function would then, for example, access the neighbor table of the MAC protocol and return the known neighbors.

3.2.5 Distributed Assertions

Distributed assertions are the basic concept in *PES* as they provide the main functionality: they describe which conditions have to hold in the network when everything is working according to the developer's intentions.
Every assertion that is evaluated in *PES* has a special timestamp connected to it; the evaluation time. This point in time is used to resolve attributes found in the assertion, which can change over time, to actual values. The assertion can contain attributes from various nodes that, together, have to satisfy the constraint described in the assertion.
The programmer places assertions in the code at the source location where they have to hold. The time when the control flow passes this location is also the time *when* the assertions have to hold and thus define the already mentioned timestamp. A slight variation of the assertion construct is a delayed assertion that behaves like a normal assertion, but is evaluated after a developer defined delay. The exact meaning is that after, for example, 1000 milliseconds, the given assertion has to hold.

Every assertion is an expression that evaluates to a Boolean value, where true means everything runs as supposed, while the value false is an indication of erroneous behavior. As the central aspect is to inspect the attributes of the nodes, it is possible to access these attributes in an assertion. An attribute is referenced by specifying the node ID that is separated using a colon from the name of the attribute that should

be accessed. If the node ID is omitted, the attribute is taken from the *this* node, that is, the node on which the assertion evaluation was encountered in the source code, also called the triggering node. Such a reference could then look like

isLeader || 100:**isLeader**

to state the assumption that the current node's *isLeader* attribute is set to true or the same attribute of the node with ID 100 is set to true.
When using delayed assertions as described above, a problem may arise in that some values in the assertion should be taken from the present, that is the time the assertion is encountered, and the others in the future, when the assertion actually has to hold. Imagine, for example, the assertion

LEADERCANDIDATE:**isLeader**

that has to hold after, let's say 1000 milliseconds, to verify that a given leader candidate has become the *leader* after this time. The value for the attribute *isLeader* is taken from the evaluation time (*now* + 1000 ms) while the *LEADERCANDIDATE* should be the one the node regards as candidate *now*. To bridge this gap evaluation constants are introduced whose value is defined at the time an assertion is encountered in the node source code. As the name suggests this value will not change over time and only holds for the evaluation of the assertion it was supplied with. Another way to think of it is that evaluation constants are simply placeholders in the assertion whose value have to be defined when the assertion evaluation is triggered. To distinguish node attributes from evaluation constants there is a ¢ prepended to the constant name, such that the above assertion correctly looks like the following snippet:

¢**LEADERCANDIDATE**:**isLeader**

To access the neighborhood of a node, there is the *hood(D)* function that makes it possible to get a set of neighbors within distance D for the *this* node. The only argument gives the distance a node might be away in number of hops. The function *nodes()* is similar but takes no argument and returns all nodes in the network. To work with these finite sets of nodes, there are quantifiers that offer the possibility to check a condition that has to hold for at least one (*exists*) or all (*all*) nodes in the set. As the quantifiers iterate over the set of nodes, the current node from the set can be accessed using the node ID *$*. An example that uses the *hood* function together with an all quantifier could look like this:

all(**hood**(1),!$:**isLeader**)

This assertion checks that none of the direct neighbors has its *isLeader* attribute set to true. To count the number of nodes in a set that fulfill a certain condition, there is the function *count* that takes as a first argument the set to count and in the second parameter, which is optional, gives the condition that has to hold for a node to be counted. The syntax of the condition is the same as for quantifiers, that is the current node can be referenced using $. An example is an assertion that checks that the number of nodes in a network that think themselves as group members is below 5:

count(**nodes**(),$:**isGroupmember**) < 5

As can be seen in the previous assertion there is also support for comparisons of Integers as well as support for basic arithmetic and logical operators. To make it possible to access the ID of a node, the keyword *id* is introduced that simply returns the ID of a node. This might be used to exclude the *this* node from a quantifier like, for example, in an assertion where a node wants to make sure that it is the only node with the *isLeader* attribute set to true.

all(**nodes**(),!$:**isLeader** || $:**id** == **id**)

To give a node application developer the possibility to adapt the assertion language to his needs, there is also a way to include user-defined functions. The developer has to implement a specific Java interface that allows querying the function name, the parameter count and types and the return type. The implementation is then registered with the evaluation system and whenever the function name appears in an assertion, the registered class will be asked to process the parameters and return the correct return value.

The assertions described so far are all assertions written by an application developer into the code of the node application to express the conditions that have to hold whenever the control flow passes these places in the code. In addition to these assertions with a fixed point in time they have to hold, there are global assertions. These assertions have to hold all the time and are not written into the application code but provided by the application developer by some other means, such that the assertion evaluator can access them. As a global assertion has neither a local *this* node nor a fixed time, when it has to be evaluated, there are some restrictions to the expressions used in them. The missing local node makes it mandatory for node attributes to always be referenced in a qualified manner, that is, specifying the node in the form *NODEID:ATTRIBUTENAME*. As the evaluation of a global assertion is never triggered explicitly evaluation constants are also missing.
All parts of an assertion expression can be found in table 3.1. A full BNF is given in appendix B.

3.3 Assertion Integration

Within the sensor node code, two main operations are needed: State publication and assertion evaluation. For this reason two constructs are provided:

SNAP is used for taking snapshots of certain state variables. The syntax looks as follows:

SNAP(*"aStateVariable[,aStateVariable2]∗"*, value1, value2, ...)

The first argument is a list of strings, naming the node's state variables that were changed, followed by a variable length list of expressions (variables, constants, ...) that provide the values for the snapshot. The semantical meaning of SNAP is that from this statement on the values of the listed attributes changed to the values supplied. Given here as an example is a snapshot that gives notification of two changed attributes.

SNAP(*"isLeader,leader"*, **is_leader**, **leader**);

Component	*Syntax*	*Semantic*
Arithmetic Operators	INTEGER [+,-,*,/] INTEGER	Usual arithmetic semantics. / means integer division.
Boolean Operators	BOOLEAN [&&, ||] BOOLEAN	Usual Boolean semantics.
Boolean Negation	! BOOLEAN	Boolean negation.
Equality	EXPRESSION == EXPRESSION	Equality, both expressions have to be of the same type.
Comparison	INTEGER [<=, >=, <, >] INTEGER	Usual comparison semantics.
Attribute	[NODE:]ATTRIBUTE	Returns the attribute value for a node. In local assertions if no node is given, it defaults to the *this* node. Every node is referenced by its ID, which is an Integer as described in section 3.2.1.
Evaluation Constant	¢CONSTANTNAME	Returns the value supplied by the triggering node. Only for local assertions
Quantifiers	exist/all(NODESET, EXPRESSION)	Returns true if at least one/all nodes in NODESET fulfill the Boolean EXPRESSION. The current can be accessed in the expression using $.
Built-in: Count	count(NODESET[,EXPRESSION])	Returns the number of nodes in NODESET. If a Boolean expression is given, the number of nodes that fulfill it is returned.
Built-in: Hood	hood(DISTANCE)	Returns the set of nodes that are within DISTANCE hops reachable from the local node.
Built-in: Nodes	nodes()	Returns a set of all known nodes.
Built-in: Max/Min	max/min(NODESET)	Return the biggest/smallest node ID in the supplied set.
Built-in: ID	id	*id* is a keyword that behaves like an attribute and returns the node ID as an Integer.
User-defined functions	FUNCTIONNAME([ARGUMENT1 [, ARGUMENTS]])	Calls a user-defined function with the given arguments. For more details see section 4.3.3.

Table 3.1: All parts of an assertion

PDA is used to trigger the evaluation of an assertion. Its syntax looks as follows:

PDA(*"Assertion"*, evaluationConstant1, evaluationConstant2, ...)

The main part of a PDA operation is the assertion that is given as a String and will be evaluated whenever the control flow passes this place. The other arguments are evaluation constants, as described previously, whose values for an evaluation are set, whenever the evaluation is triggered. An example is the evaluation of an assertion which checks that all members, nodes with the attribute *isGroupMember*, know the current leader, that is, have their attribute *leader* set to the same value as is supplied for the evaluation constant ¢LEADER.

PDA(*"all(nodes(),!$:isGroupMember || $:leader == ¢LEADER"*, **leader**);

An extension to assertions that have to hold at the exact time the control flow passes the assertion are delayed assertions, as described above. These assertions additionally carry a delay that is added to the current time and then gives the time an assertion has to hold. The syntax looks quite similar to the instant assertions except that the delay in milliseconds is supplied as the first argument and the keyword is not PDA but D_PDA.
Delayed assertions have the advantage that they allow the formulation of an assumption about the future network state that will hold after some time - if everything runs as expected - with no more effort than using instant assertions. Using only assertions that are evaluated immediately would require the node to contain a structure like a timer that then checks the condition using a normal assertion after the delay. As this timer has to be configured by the application developer, this has too much impact on the node code and is, therefore, unacceptable. Delayed assertions allow this postponed evaluation at no cost beyond the fact that the developer has to be sure that the network state will develop in the form described in the delayed assertion and within the time constraint given by the delay. Otherwise, a spurious assertion violation will be reported.

3.4 Assertion Evaluation

The assertion evaluation component needs access to the network state for the evaluation of the assertions. Therefore, a network model is introduced that provides this service. It collects all snapshots and processes them such that the information can be accessed in a convenient way. This is the reason for splitting *PES*'s backend into two main components: one to keep a history of all gathered node states and one that effectively evaluates the assertions. The two modules are described in the following two sections, starting with the node state history, called the network model.
To integrate the assertion evaluation with the network, a kind of connector is also needed which records the messages via radio communication and transfers them to the appropriate component, that is, it directs all snapshots to the node state history and passes all evaluation requests to the evaluator. Figure 3.3 shows the basic design for the *PES* backend.

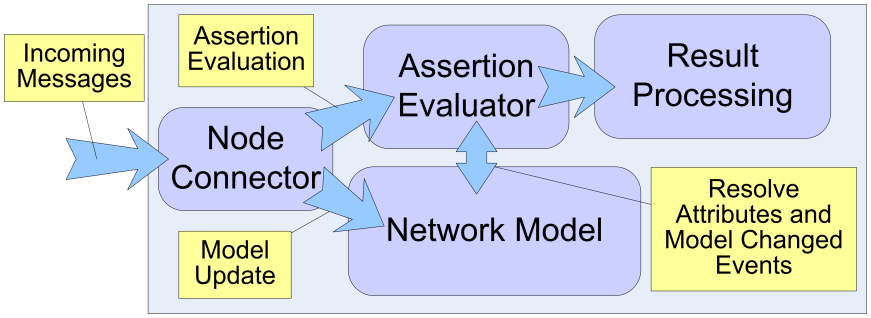

Figure 3.3: The basic layout used for collecting and evaluating the network state.

3.4.1 Node Connector

As already described, SNIF is used to collect the messages that hold the snapshots and the assertions defined in the node code. The connector implements the needed interfaces to be integrated with SNIF and then receives a stream of messages as collected by SNIF. The messages in the incoming stream still need to be checked for duplicates, correct message types and for integrity. These tasks are accomplished by the connector. What then remains are messages that are either snapshots or assertions and have not been processed before.
For snapshots, all attributes are extracted and, together with information about the originating node and the timestamp, passed to the network model. For assertions, the evaluation constants are retrieved from the message and the assertion evaluator is instructed to evaluate the assertion using the extracted evaluation constants.

3.4.2 Network Model

The network model's main purpose is to manage the information collected by snapshots. The network's evolving state is recorded gradually in order to have not only the most recent state, but also the whole history of it. The network model then makes this information accessible to the *PES* assertion evaluation.
The network state can be divided into two different parts; namely the state of the nodes with regard to their attributes, and the connectivity between the nodes. This distinction is made to allow these two types of state to be collected in different ways and thus independently of each other.

For a single node, the internal state is divided into different attributes, whose name is then used to access them. As the value of an attribute is assumed to change over time, a history is kept, in order that the system is able to respond to queries about the value of a given attribute at any given time. For every snapshot that updates the value of an attribute, the new attribute value, together with the timestamp of the change, is stored in the attribute. This basic and slightly idealistic view is expressed in figure 3.4.
Unfortunately there are several limitations that lead to uncertainties when trying to resolve an attribute value, which necessitates a more complex interface. SNIF, which is used to collect snapshots, orders the incoming messages chronologically and assigns a timestamp to each message. This timestamp, however, is not fully accurate and

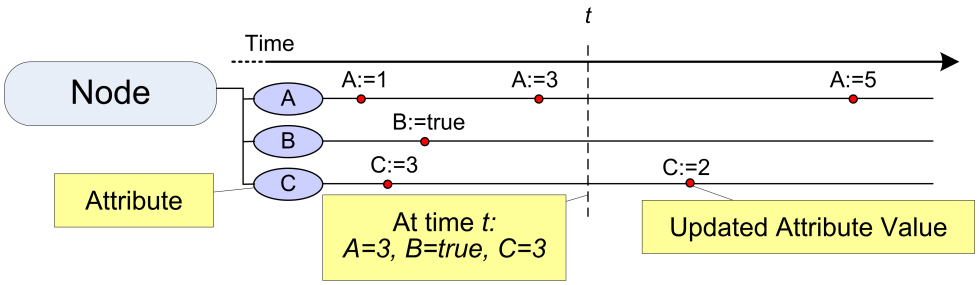

Figure 3.4: The network model as it should be: No lost messages, accurate message serialization.

consequently the messages' timestamp might be off by a few milliseconds.

When a client requests the value of an attribute at a given time t, the network model associates a level of confidence with the return value based on some inner rules and taking the possible error sources into account. The confidence is split into three states: *possible*, *verified*, or *unknown*. A *verified* value is stable in the sense that it will not change and is regarded as correct under some assumptions, mainly that the error in the message timestamp is within a certain range. A *possible* value on the other hand is one, where a value could be inferred, but the model is not or not yet sure if the value is correct. Of course, a value can also be unknown for various reasons, such as for a node that never sent a snapshot for an attribute, or a node whose reboot was detected.
To account for the imperfect message serialization, a value that lies too close to an attribute value change will always be *possible* as it is not definitely possible to tell whether the attribute value changed before the request or after the request. Figure 3.5 shows how the value around an attribute value change is regarded as uncertain.

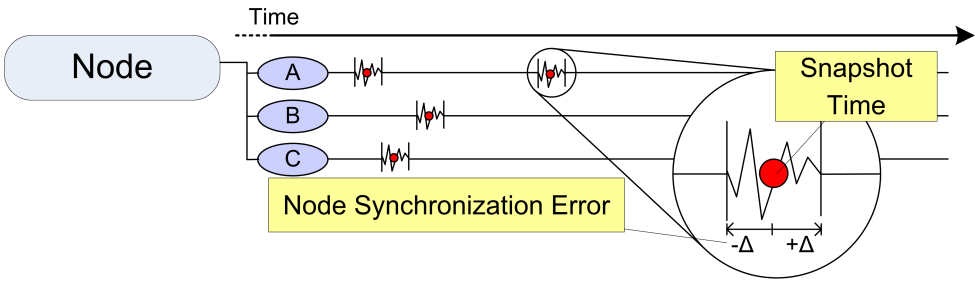

Figure 3.5: A network model that accounts for nodes with not completely synchronized clocks.

Another source of errors can be found in missing attribute updates due to message loss, that is, snapshot messages that got lost because of packet collisions, checksum errors, or external influences. When an attribute value change goes unnoticed, the system might return a wrong attribute value without being aware of this. To prevent such errors, a sequence number is integrated into each message sent on behalf of *PES* on a per-node base. This allows not only detection of lost messages, but also node reboots in the same way it is accomplished in SNIF, namely by detecting a sequence

number that is far away from the next expected sequence number, which suggests a reinitialization of the sequence number counter that is only done on reboot. When a lost message is detected for a node, all attribute values of this node have to be treated as possibly changed, as the lost message could have changed all attributes. These attribute values stay *possible* until a new snapshot updates the value. A slightly more serious error is a node reboot. In this case, all detected attributes have their state changed to *unknown* since attribute values before reboot are most likely unrelated to their values after reboot. To prevent such an unknown state, a node is expected to always post a snapshot to initialize the state of all node attributes when starting up. Figure 3.6 illustrates a lost message and, how, after a reboot or after a lost message, all attributes have to become either *possible* or *unknown*.

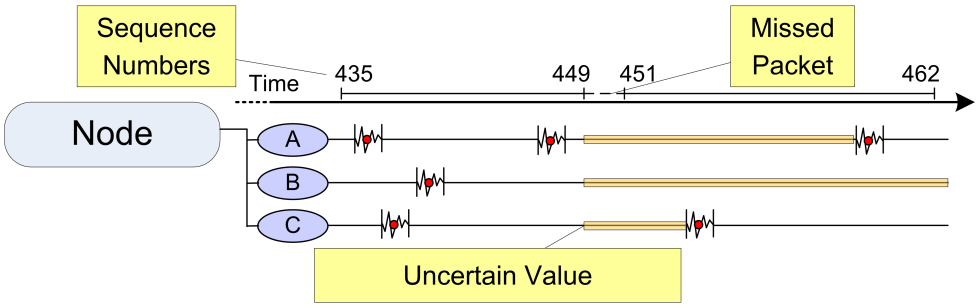

Figure 3.6: A network model that detects and handles lost attribute changes.

When it comes to initializing or reinitializing the node state, periodic updates as described in section 3.2.4 are very important. They provide the full node state to make up for lost messages and allow the network model to mark nodes that did not submit any values as inactive.

The interface to the network model for querying values of attributes contains two methods of the form *getVerifiedValue(node, attribute, time)* as well as *getPossibleValue(node, attribute, time)*.
To support assertion evaluations that are triggered by a changing attribute or a changing node connectivity, an event mechanism is needed that publishes changed attributes and newly detected or lost nodes. Every interested component can then register itself with the network model and react appropriately to these events. This infrastructure is mainly intended to be used when evaluating global assertions, but is not limited to it. Another possible use is an attribute viewer that visualizes the network, the nodes' states and their changes over the time.

The connectivity between the nodes is treated separately to allow different components to supply this service. Next to a connectivity model which receives its content from snapshots, there could also be a connectivity model which integrates SNIF's capability to deduce a node's neighborhood by analyzing to whom it sends messages. The connectivity model is quite simple and allows queries of the form *getNeighbors(node,time)*. Although a neighbor's hood can be seen as yet another attribute, it does not distinguish *possible* values from *verified* ones. This decision was taken because the *hood* of a node is not expected to change often, which reduces the probability of errors enough that the advantage of easier integration of other connec-

tivity monitoring technologies exceeds the disadvantages.
As seen from any client external to the network model, the connectivity model is contained within the network model, such that all model access is done via the network model. Calls to the connectivity model will be redirected if needed and thus the above mentioned methods also belong to the network model interface.
Next to an interface that allows access to the node state some interfaces are also needed to describe how the network and connectivity models consume new snapshot information. A class implementing those interfaces takes an attribute's value, its name, and the node the snapshot comes from, as well as a timestamp and stores the information appropriately in its inner representation. A similar interface is needed for storing connectivity model updates. It takes either a set of nodes together with a timestamp for reporting the neighborhood at this specific time or two sets, one with removed and one with added neighbors, to report a changed neighborhood at a given time.

3.4.3 Assertion Evaluator

The assertion evaluator is the component that does the actual evaluation of assertions. Given an assertion and an evaluation timestamp, and - if available - the local node, it evaluates the assertion. Every assertion evaluation has an outcome that reflects the evaluation result and a status that gives the reliability of the outcome. The outcome might be *unknown*, *success* or *fail*, the status is one of *not evaluated*, *tentative*, *finished* and *dropped*.
Depending on the involved nodes and attributes, some values might not be available or definitive. When an *unknown* value is used during the evaluation of an assertion, the outcome of the whole assertion is *unknown* as well and the status stays *not evaluated*. When a *possible* attribute value is used, the assertion is nevertheless being evaluated fully to give a first impression on the network state. The evaluation's status then becomes *tentative* to indicate the temporary nature of the assertion's outcome. Non-evaluated or not fully evaluated assertions are then re-evaluated after some time and only left unevaluated after several retries. The assertion evaluation status then becomes *dropped*. In case of an assertion not being evaluated definitely, the next evaluation try is scheduled after some seconds, because some values that were *unknown* or only *possible* might have become *verified* in the meantime. For every re-evaluation the sleep interval is doubled until the assertion is evaluated successfully or a maximum number of retries is reached and the evaluation *dropped*. It is important to see the difference between the evaluation time, that is the time the assertion should hold and the point in time the assertion is evaluated.

Whenever an evaluation is started or its outcome changed, an event is sent out to inform interested listeners, like a GUI or a logging client about the changed evaluation result. To be of use, an evaluation result contains not only the assertion text and the evaluation result, but also some steps in between, like values for the different attributes used in the evaluation. Another feature that is not vital but nevertheless desirable, is the ability to provide for each assertion the name of the file and the line it comes from. This allows easier error diagnostics and thus better debugging.
Figure 3.7 illustrates the evaluation of the assertion

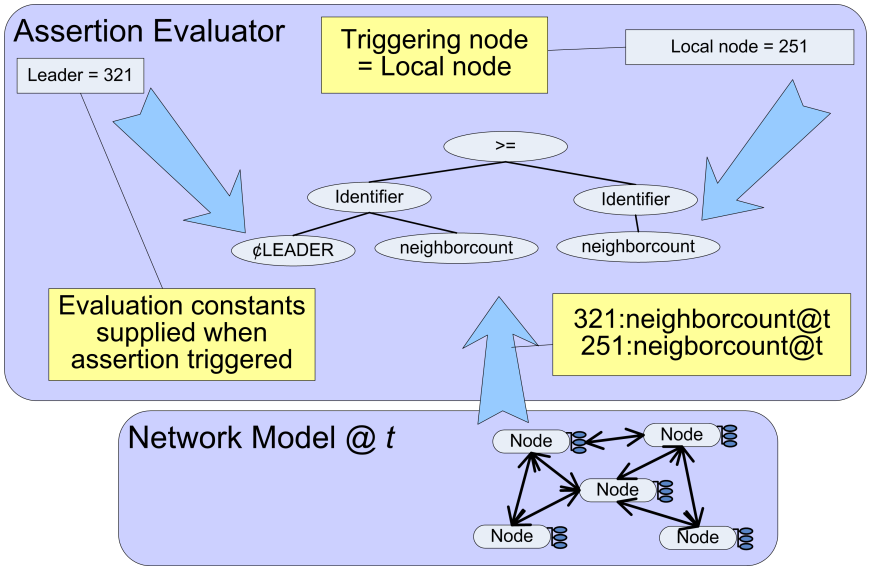

Figure 3.7: An assertion evaluator that evaluates an assertion for a given time. It requests an attribute from two different nodes whereas one is the *this* node whose ID is supplied implicitly.

¢LEADER:neighborcount >= neighborcount

that is encountered somewhere in the code as .

PDA(*"¢LEADER:neighborcount >= neighborcount"*, **leader**);

The node, where this assertion is encountered, assumes that - for whatever reason - the attribute *neighborcount* for it is smaller or equal to the one of the node it refers to as *leader*. In this context the *leader* is not an attribute of a node, but a value the node has calculated and therefore is inserted in the assertion as an evaluation constant. When the assertion is being evaluated by the assertion evaluator, it starts gathering all required values which are:

ID of the local node This value is given to the evaluator when the assertion is triggered by the node connector that extracts it from the received message.

¢LEADER The evaluation constants, including ¢LEADER, are sent together with the request to evaluate the assertion and therefore were also supplied when the connector forwarded the request to the assertion evaluator.

Neighborcount Neighborcount is an attribute and therefore its value at evaluation time t is retrieved from the network model.

The assertion evaluator also handles the evaluation of the global assertions. The idea is that whenever the network changes, it must be ensured that the global assertions still hold. To do this, the evaluator analyzes all global assertions and creates a map that assigns to every attribute a list of global assertions that depend on those attributes. Whenever an attribute changes, these assertions are evaluated.
Unfortunately the concept of *possible* values brings in a problem since the evaluation

time for such a global assertion is the same as the time the attribute changed. The network model, however, will only return a *possible* value for the initially triggering attribute that will then lead to the assertion first receiving the status *tentative* and finally *dropped* as not all values are *verified*.
A possible solution to this problem is that the evaluation time of an assertion is moved forward in time, until all values are settled and no *possible* values result from the fact that the evaluation time is too close to the time when the attribute value was received.

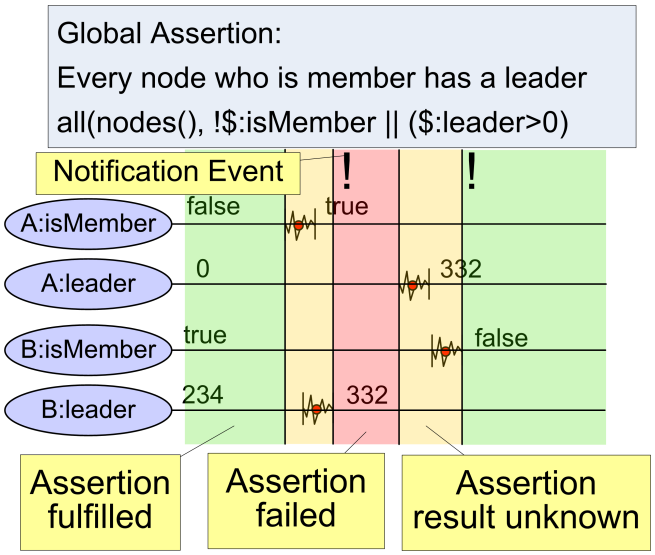

Figure 3.8: A global assertion and its evaluation result how it changes over the time.

Figure 3.8 illustrates this idea. There the global assertion .

all(**nodes**(), !$:**isMember** || ($:**leader**>0))

is given in the form of its abstract syntax tree, which checks that all nodes that are members of a group know the leader, assuming that a leader equal to zero indicates no known leader.
In the beginning, the assertion is fulfilled, as only node B is a member and has a leader > 0. Then suddenly at time t_1, A becomes a member as well, therefore the global assertion is queued to be processed after the expected node synchronization error Δ at $t_2 = t_1 + \Delta$. But at t_2 the attribute *leader* has changed its value just recently, at t_3, and therefore the evaluation is again postponed until also this value is settled. The new envisaged evaluation time now is $t_4 = t_3 + \Delta$. At this time the assertion can be evaluated and results in a fail, which is then reported via the normal event processing mechanism provided by the assertion evaluator. The same happens a bit later when the values of *leader* are updated as well.
This solution introduces time spans when the system is not sure about the outcome of a global assertion, but on the other hand gives a way to have global assertions checked in a reliable way at other times. Due to a lack of time, this solution was not implemented and therefore global assertions will end with a status *dropped* because of *possible* attribute values.

4 Implementation

As discussed in the chapter *Design* (section 3), the software consists of two parts. One to publish the nodes' states (executed on the sensor nodes) and one to collect, access and analyze the different node variables at a central point (executed on a backend system). This chapter gives the reader some insight into the details of the two implementations in order to understand the design decisions and to assist him or her to find his or her way into the source code. In the beginning of this chapter, the preprocessor is described followed by the implementation details for the two software parts, written in C and Java.

4.1 Preprocessor

When assertions and snapshots are added to a sensor node application, they are for the most part embedded in the application code. The preprocessor takes care of extracting the assertions from the C code and makes them available for the assertion evaluation software. Besides this, the preprocessor provides data structures that will enable the *PES* sensor node connector to use less resources at runtime, and thus reduce the impact *PES* has on the node application. This section describes in detail the steps that are taken by the preprocessor.

4.1.1 Preparation

Because in the preprocessing stage the source code of the original sensor network application is modified, all files found in the project directory of the node application are copied to a new folder with the same parent as the original folder, but with an *_build* appended to its name. All files with the extension *.c* are processed to retrieve the snapshots and assertions and to prepare the node code for the later execution.

4.1.2 Assertion Extraction

Instead of simply copying the C files, they are read line by line and a regular expression is used to find all Strings of the form

```
["PDA" | "SNAP" | "D_PDA"] "(" (<DELAY>",")? <ASSERTION/SNAPSHOT>
    ("," <EXPRESSION)* ")")
```

Every assertion and snapshot found is assigned a unique ID, which then is used to replace the `<ASSERTION/SNAPSHOT>` part in the found expression. This ID will then be used to access the data structures described later and to communicate with the backend system in a more efficient way, as only an ID, instead of the whole text, has

to be transferred.

The retrieved assertions and snapshots are written to a file, one element per line. This file is later used by the assertion evaluator to map IDs to assertions. Besides the assertion and snapshot text, the file also contains the name of the file the assertion was found in, the line it comes from and the type - PDA or SNAP - of this line. The file has the format given below.

```
<FILE> <LINE> <TYPE> <ID> <ASSERTIONTEXT/SNAPSHOTTEXT>
```

In addition to the changed parameter in the SNAP and PDA macros, also `#define PDA_RESOLVED_` is inserted at the beginning of the file, which causes the macro definitions to accept the newly inserted integer arguments, instead of the replaced strings.

4.1.3 Type derivation

With all assertions extracted, the preprocessor tries to assign a type to every attribute and evaluation constant. This type information is needed on the evaluator and the sensor node to correctly encode and decode the messages. It is also used to check the type safety of assertions.
To derive the types, every attribute and evaluation constant is assigned a set of possible types, initially all available types. Now all assertions are inspected to see at which positions in an expression an attribute is used, and all types that are not allowed in such a place are removed from the list of possible types. Ideally only a single type remains when all assertions are processed. If it is not possible to derive the type, it has to be specified by the user in a file which has, on every line, the type and the name of an attribute or evaluation constant in the following format:

```
<TYPE> <SYMBOL>
(For Evaluation Constants: <SYMBOL> =  <ID>.<SYMBOL>)
```

Where `<TYPE>` is one of `Boolean` or `Integer`. All identifiers for evaluation constants and attributes are then written to a final file that has the same format as the one given further up.

4.1.4 Node Preparation

Now that the assertions and snapshots are replaced by IDs between 0 to n-1, where n is the number of assertions and snapshots, the essential information from the assertions and snapshots, such as the number of parameters and their type, has to be made available to the *PES* sensor node code. Everything needed is written to the file *pdatable.h*, which is added to the build directory and thus included into the build process.
For every assertion and every snapshot there is, as already mentioned, the number and the types of the parameters required to write the attributes or evaluation constants in the correct format into the message that will be broadcast. This information is stored in the array `__pda_format` at the index with the same value as the ID for the SNAP or PDA. For every occurrence of SNAP and PDA, there is a String of the form `[h]-[d|b]*` in the array, which will later be referred to as *packet information*, as

it contains everything needed to prepare the packet. All values before the dash are options for the packet. Only one is available now, the `h` means that the *hood* is sent with this packet, if hood sending is enabled. After the dash there is a list of types, with the meaning the i-th char gives the type of the i-th attribute value in a snapshot or the i-th evaluation constant in an assertion. The `d` and the `b` stand for the types Integer and Boolean respectively. This is everything needed to prepare a packet.

For use in *periodic updates*, every node also caches the values published via snapshots previously. The attributes, like the assertions, are numbered from 0 to m (where m is the number of attributes) and the attribute ID is used to access the arrays that contain information about the attributes. The header file *pdatable.h* contains an array, with as many entries as there are attributes. The array is called `__pda_symbol_cache` and contains pointers, which, when the cache is initialized, point to the memory region that holds the cached value. An array of the same form, with the name `__pda_symbols` contains the types for all the attributes.
To assign the values contained in a snapshot to the different attributes, a mapping from snapshot to contained attributes is needed. This is done using the `struct` given in listing 4.1, which contains the number of attributes in a snapshot, together with an array of this size, which is filled with the IDs of the updated attributes. An example of a *pdatable.h* file can be found in listing 4.2.

```
typedef struct snap_to_symbol_s {
  u_char size;
  u_int snaps[];
} snap_to_symbol_t;
```

Listing 4.1: The `struct` used to store snapshot to attribute mappings.

```
/* SNAPSHOTS & PDAS
  0 SNAP isGroupmember,isLeader,leader
  1 SNAP leader;
  2 PDA leader:isLeader
  3 PDA ¢LEADERCANDIDATE:isGroupmember
*/
const u_int __pda_symbol_count = 3;
const u_int __pda_assertion_count = 4;
// The symbols are: isGroupmember,isLeader,leader
u_char* __pda_symbols[3] = {"b", "b", "d"};
// The attribute cache initalized with NULL pointers.
u_char* __pda_symbol_cache[3] = {NULL, NULL, NULL};
// A mapping that assigns
// Snapshot 0 has 3 attributes, in the order 1,2,3
static snap_to_symbol_t snap_0 = {3, {1,2,3}};
// Snapshot 1 has a single symbol, the symbol with ID 3
static snap_to_symbol_t snap_1 = {1, {3}};
// Mapping from symbol ID to symbol information
snap_to_symbol_t* snap_to_symbols[] = {&snap_0, &snap_1, NULL, NULL};
// The different messages
u_char* __pda_format[4]= {"-bbd", "-d", "-", "-d"};
```

Listing 4.2: An example for a pdatable.h file.

File/Folder (default)	*Description*
Project folder	The project folder that contains all application source files. The whole folder is copied and C files filtered. The build folder has the same name with a *_build* appended.
Assertions file (*assertions.txt*)	The assertions file contains all assertions and snapshots together with their preprocessor assigned IDs and some additional info such as line numbers.
Type file (*types.txt*)	A simple list of all node attributes, evaluation constants and their types
pdatable.h	A header file included in the node application that carries all resources needed on the node, listing 4.2 on page 23 shows an example.
User-supplied type file (*usertypes.txt*)	This file is used in the step of type deduction and helps to fix the type for some attributes and evaluation constants that cannot be derived automatically.

Table 4.1: Summary of all files used or generated in the preprocessing step.

4.1.5 Compilation

When the node application code has been processed and the necessary helper files created, the application is built, using *make clean all*, and can then be uploaded to the node by issuing *make burn btnode3*.

Table 4.1.5 lists all files and folders that are used or created during the preprocessing step.

4.2 Node Integration

To ease the integration with present and future wireless sensor network applications, the node connector is realized as a C header file, with the name *pes.h*, that currently has to be copied into the application directory, but could also be moved to the `include` directory of the BTnut distribution.
The main interaction between application developer and *PES*, takes place via two macros: `PDA` for evaluating assertions and `SNAP` for taking snapshots. These macros are then translated into function calls, that publish the attribute values of the nodes and that instruct *PES* to evaluate an assertion.
Besides these two macros, there also exist some functions to configure the PDAs which are discussed later in this section.

4.2.1 `PDA` and `SNAP`

The two macros `PDA` and `SNAP` are translated into debug messages, unless the `#define PDAS_RESOLVED_` is set by the preprocessor. After the preprocessing step, these macros both call the function `eval_pda`, which takes at least three arguments, the type of the call (PDA or SNAP), the ID of the assertion or snapshot, the evaluation delay

(only used for assertions) and then a variably sized list of arguments, which provide additional values either for the evaluation of the assertion or as values for the node attributes contained in the snapshot.
Both macros use the same function since the functionality required is nearly the same: For publishing a snapshot, a message has to be broadcast which contains the ID of the snapshot and a list of values that were assigned to the attributes. When a `PDA` macro is encountered in the code, the *PES* system is requested to evaluate an assertion that is given by a single ID. Together with the assertion ID, a list of evaluation constants are also supplied that have to be included in the broadcast message as well. An overview over the evaluation process is given in figure 4.1. The process described here starts with the function `eval_pda` and ends with the delivery of a message.

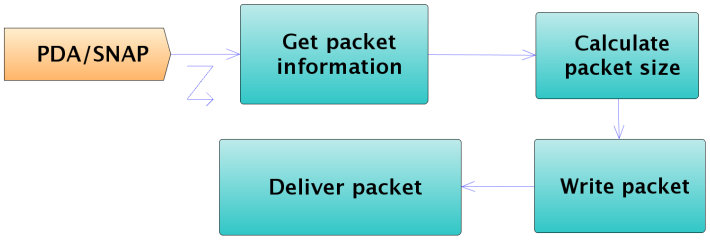

Figure 4.1: Overview of the general evaluation process for SNAP and PDA.

First the *packet information* provided by the preprocessor is requested using the ID supplied by the macro. A detailed description of all resources created by the preprocessor can be in found in section 4.1.4. Here it is sufficient to know that the *packet information* holds a flag to tell whether the *hood* has to be sent and the number and types of the parameters, which are included in this message. Using this information, the packet size is calculated. Figure 4.2 shows which factors are taken into account when calculating the packet size.

The neighborhood of a node is needed twice. First while calculating the size of the packet that has to be allocated, and secondly to effectively write the neighborhood to the message. As in all concurrent systems, the *hood* could possibly change during these two steps. Therefore, when calculating the size of the hood, the message content that will be added to the message is already written into a buffer and later copied into the message. Obviously this needs careful locking between the calculation of the *hood* size and the writing to the message in order to prevent other threads from overwriting this buffer. Further information can be found in the source code. The rest of the dynamic message size is simply the sum of the sizes of the parameter types.

Figure 4.3 shows the details about the writing process. It is basically the same as for calculating the size, except that all the data is written and the flags for the different packet options have to be set. In the end the message is queued for delivery.

To deliver a message, it is passed to the function `deliver_pda_values`, that takes all packet information and guarantees that the message will be sent eventually. In the current implementation, a message is first queued and sent asynchronously by

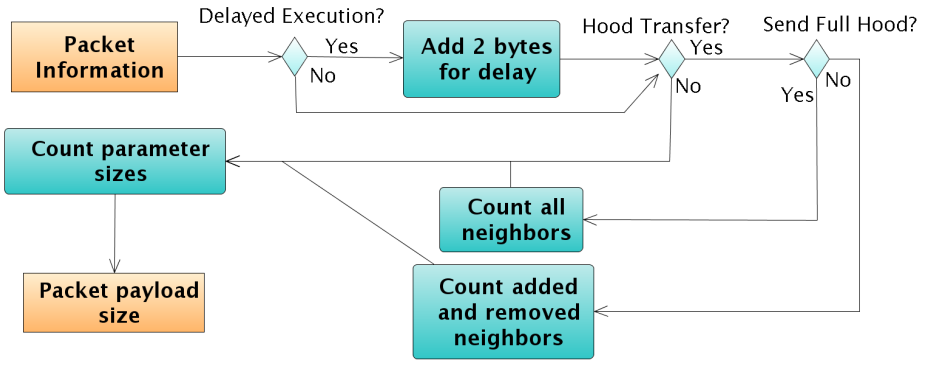

Figure 4.2: Detailed actions for calculating the packet size.

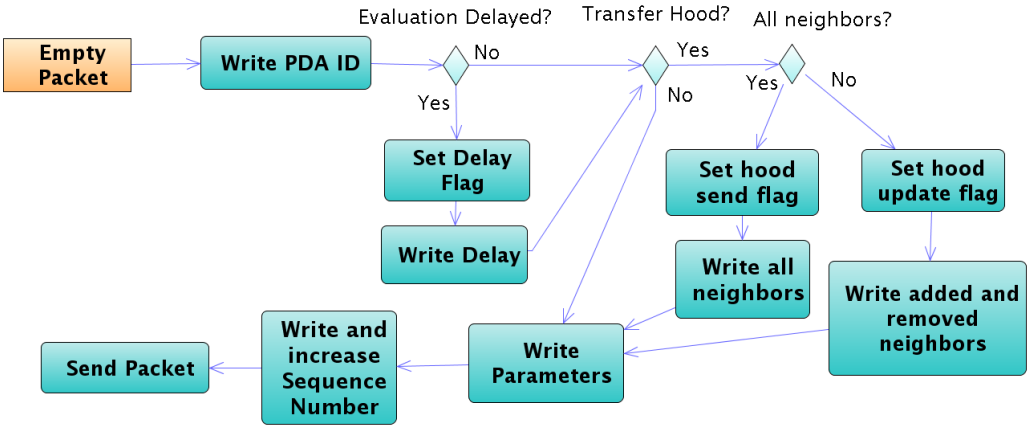

Figure 4.3: Detailed actions for writing and delivering a snapshot or an assertion evaluation.

a dedicated thread. This is done to have a minimal impact on the timing of the application code. To counteract congestion during phases of activity, sending is also delayed for a random time, up to one second, as it is to be expected that multiple nodes will trigger evaluation of assertions and change their state at virtually the same time, when some external event triggers the nodes to act. Shortly before the message is sent, the current node timestamp is written to the message by the MAC protocol. This timestamp is used, together with a timestamp taken immediately after the SNAP or PDA invocation, to calculate the delay that was introduced by the node application. This delay will be subtracted from the message's receive timestamp to improve the accuracy of the network model. Figure 4.4 shows how the difference between the evaluation timestamp t_1 and the send timestamp t_2 are used to improve the accuracy of the receive timestamp. The delays that make up the difference between t_1 and t_2 are the sum of artificial delays inserted by *PES*, the processing time of the operating system and delays imposed by the MAC protocol, which has to await a slot to send

the message. The global clock in Figure 4.4 is provided through synchronization of the deployment support network of SNIF.

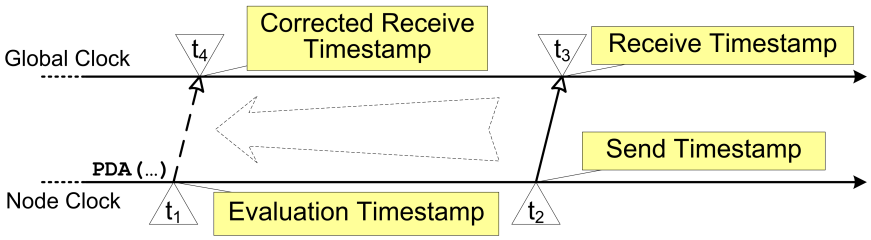

Figure 4.4: The different timestamps and how they are used to improve the accuracy of the evaluation time.

4.2.2 Periodic Updates

As described in the design chapter in section 3.2.4, some functionality is provided in order to publish the full node state in order to compensate lost SNAP messages. This is done in regular, user-defined intervals to synchronize the collected state with the evaluator. A periodic update contains values for all attributes that are taken from the attribute cache, which is described in section 4.1.4. All attribute values are written to this snapshot message in order of increasing attribute IDs. To identify this snapshot - for which no assertion/snapshot ID is present - a special flag is set, which indicates the packet structure and content. The rest of the delivery process is done via the `deliver_pda_values` function, as described above.
To ensure that the periodic updates do not interfere with the rest of the node application, a dedicated thread is present that sleeps for most of the time and checks at regular intervals whether anything was broadcast on behalf of *PES*. If not, a periodic update is sent. This interval can also be configured at runtime using the `set_periodic_update_interval` method.

4.2.3 Packet Format

Message 4.2.3 shows the basic structure of a PDA packet. It contains the PDA ID and a sequence number per node to detect lost messages and node reboots. There is also a timestamp, which is taken at the time the PDA or SNAP macro was invoked. Using this timestamp and the send timestamp, the delay can be calculated, which is then used to correct the SNIF-provided timestamp. Furthermore, there are the flags which are stored in a *flags* byte. The rest of the packet consists of dynamic content that is described below. The packet is written in network byte order, that is, big endian. Although there is no limit to the packet size by design, there is a limit in SNIF for the default maximum packet size of 100 bytes (which could be increased), that then also constrains the maximum size for the *PES* broadcasts.

There are different factors that affect the content of a packet. The dynamic content can be inferred by inspecting the flags (described below) and the PDA ID. It is possible to trigger the evaluation of an assertion with some delay to indicate that - if everything ran correctly - some condition will hold after X ms. If such a delay exists,

Base	0	1	2	3	4	5	6	7
0x00	PDA ID		Seq #		Timestamp			
0x08	Flags	Dynamic Content						

Packet 4.1: Basic PDA packet structure.

it is written as the first integer in the dynamic section and the flag *Delayed* is set.

If the *hood send* flag is set, the next part of the message contains a set of nodes that are in the neighborhood of the sending node. For the *hood update* flag it is basically the same, except that two sets are sent; the first one contains all nodes that were added and the second one all nodes that were removed since the last sent *hood*.
The last part consists of the parameters that are shipped with the message. The number and types of the parameters are given by the PDA ID and loaded from the preprocessor supplied *pdatable.h*.

Flags

Flags are stored in the message in the *flag* byte. The different flags are all powers of two and can therefore be summed up to set multiple flags.

Hood send (0x1) Defines that along with this message a full *hood* set is sent, that is, all known neighbors of the sending node.

Hood update (0x2) Defines that along with this message, a diff is sent from the last sent *hood* to the current.

Delayed (0x4) Informs the receiver about a delay that was added in front of the messages dynamic part.

Periodic Update (0x8) This indicates a periodic update packet that contains the full node state.

Types

Currently there are two possible types for attributes and evaluation constants. Data sets are only supported for node IDs as they appear in a *hood* set.

Integer Represented by a 2 byte unsigned integer.

Boolean Represented by an unsigned char with 0 meaning false and non zero meaning true.

Integer Set An integer set is the type used for hood sets. Integer sets (sets in general) consist of a 1 byte unsigned integer representing the size of the set, followed by n times the size of the base type, where n is the number of elements in a set. Although this restricts sets to a size of 255 elements, this should not be a limitation as most sensor nodes already are memory constrained and therefore no bigger data structures are to be expected.

4.2.4 API

Besides the previously mentioned PDA and SNAP, there are also other interaction and configuration possibilities when using the PDA header file. All functions and macros intended for use by the developers of the node application are given below.

PDA(String, ...) Triggers the evaluation of a PDA. The supplied arguments are used to substitute evaluation constants in the assertion.

D_PDA(Delay, String, ...) The same as PDA except that the given delay is added to the timestamp to have the assertion evaluated later.

SNAP(String, ...) Sends a snapshot for the listed node attributes. The following arguments are used to set the value of the attributes.

set/is_pda_enabled(boolean) Enables a user to switch the PDA evaluation on and off during runtime.

set/is_hood_transfer(boolean) Enables the user to change whether the *hood* is transferred with messages or not. This only applies to periodic updates and snapshots that explicitly transfer the node's neighborhood.

set/get_periodic_interval(long) Configures the interval, after which periodic updates are sent. This only has an influence when PDAs are enabled. The value should be bigger than 1000 ms, as a smaller value only leads to more collisions and thus worse results.

pda_init(hood_size_fkt, hood_fkt) Initialization function. It has to be called at the start of the application. The two arguments are both function pointers. The first one points to a function without argument, which returns the number of neighbors. The second one takes one parameter, namely the index of the neighbor from 0 to the number of nodes - 1, and returns the ID of this node. This way the PDA infrastructure supports different platforms which may handle the network neighborhood in different ways.

4.3 Assertion Evaluation

The assertion evaluation application is divided as suggested in the design section in figure 3.7. There is an adapter that takes the incoming packets and directs snapshots to the network model, and assertions that should be evaluated to the evaluator.

4.3.1 Packet Parsing

When initializing the system, SNIF is provided with a description of the packet format that then enables the sniffer to capture the packets and forward them to a packet sink. Packet parsing is the first step for incoming packets in assertion evaluation. Its central class is `PDASink`. This class implements `AbstractSink<PacketTuple>` and is therefore the entrypoint for all packets captured via SNIF.

The *PES* packet, as described in section 4.2.3, is read through an interface called

`PacketReader`, which grants access to all parts of a packet in an abstract way. There is currently only one implementation with the name `SimplePacketReader`, that can be found as internal class of `PDASink`. For each packet, a `PacketReader` instance is created, that is then used to access the packet contents. Upon instantiation, the reader checks the CRC of the packet, to ensure its integrity. If the packet is not valid, it is dropped, that is, it is simply ignored. Dropped packets are not unusual and are expected to be encountered during regular operation due to the high probability of bit errors and collisions.
Once the packet's CRC is checked, its type is inspected to ensure the correct packet type. Valid types are stored in the class as constants and are currently 0x12 for snapshots and 0x11 for assertions. The next thing to check is the sequence number. As there is a separate sequence number for each node, the sender address also has to be retrieved from the message. `PDASink` checks with the network model that the message is not a duplicate, which would be ignored.

If a set of neighboring nodes is sent with the message, this is indicated by the message flags and the updated *hood* is retrieved and stored in the connectivity model. Now, based on the ID of the message, a list with the expected number and the types of values sent with the messages is loaded, and the values are extracted from the packet. If the message was a snapshot, these values are attributes of this node and have to be forwarded to the network model. Otherwise the message is a triggered assertion, for which the supplied values are evaluation constants, which are stored in a map which is passed to the assertion evaluator, together with the ID of the assertion that has to be evaluated.

4.3.2 Network Model

The network model represents an abstract view of those aspects of the sensor network that are relevant for assertion evaluation. The interface of the network model can be found in listing 4.3. The first three methods are the most commonly used ones, while the others are only used during start-up.

```
package ma.model;
public interface NetworkModel {
  /**
   * Get the attribute symbol for node
   */
  Attribute getAttribute(int node, String symbol);

  /**
   * Get the neigbhors symbol at time.
   */
  Set<Integer> getNeighbors(int node, long time);

  /**
   * Get all nodes at time.
   */
  Set<Integer> getAllNodes(long time);

  /**
   * Register a listener that gets an event whenever an
   * attribute changes or a node appears.
```

```
     */
    void registerNetworkModelChangeListener
          (NetworkModelChangeListener listener);

    /**
     * Unregister a listener.
     */
    void unregisterNetworkModelChangeListener
          (NetworkModelChangeListener listener);

    /**
     * Set a new connectivity model
     */
    void setConnectivityModel(ConnectivityModel connectivityModel);
}
```

Listing 4.3: The `NetworkModel` interface

The `Attribute` returned by the `getAttribute` method represents an attribute as found on a node. It encapsulates the attribute's history and therefore allows queries for the attribute value at different times.

The implementation of this interface written in the context of this thesis is named `AdaptingNetworkModel` and provides a model that dynamically creates and deletes nodes, when no messages are received from them for some time. It also assumes that when a difference in sequence numbers above a given threshold is detected, the node has rebooted, as it is unlikely to miss a certain number of messages in a row. Therefore the state of such a node is cleared, as if the node hasn't been seen yet. The class contains several inner classes, most notably two threads of which one carries out notifications for newly detected nodes and changed attributes and one which removes nodes that haven't been sending messages for some time.

Another part of the network model is the connectivity model, which represents the network connectivity between the nodes. The connectivity model is contained in the network model and thus the methods for querying the connectivity between nodes are available in both interfaces. The associated interface is called `ConnectivityModel`, the main implementation is `NetworkModelConnectivityModel`. It allows queries about connectivity between two specific nodes, a node's neighborhood and all known nodes. Listing 4.4 shows the interface. Its realization is named this way as the connectivity model uses the same node information as the network model for determining if a node is visible. There is also another implementation of the `ConnectivityModel` that is currently not used. It is the the `StaticConnectivityModel` that has a fixed set of nodes with a constant connectivity between the nodes.

Next to the models, there are also two interfaces which reflect the model's consumer nature, namely the ability to accept connectivity and node state information and store it somehow. These consumer interfaces are implemented by the `AdaptingNetworkModel` and the `NetworkModelConnectivityModel` and correspond strongly with the respective *model* interfaces; one interface to absorb state information - the consumer interface - and one to make the prepared state information accessible - the model interface. These two can be seen in the listings 4.5 and 4.6. The `NodeStateConsumer` interface also implements the interface `SequenceNumberConsumer` that offers the pos-

```java
package ma.model;
public interface ConnectivityModel {
  /**
   * Are/were a and b connected at time?
   */
  boolean isConnected(int a, int b, long time);

  /**
   * Get all neigbhors for node at time.
   */
  Set<Integer> getNeighbors(int node, long time);

  /**
   * Get all nodes in the network at time.
   */
  Set<Integer> getAllNodes(long time);
}
```

Listing 4.4: The `NodeConnectivityModel` interface

```java
package ma.model;
public interface NodeStateConsumer extends SequenceNumberController {
  /**
   * Report the state of an attribute.
   */
  void reportAttribute(AttributeValue value, String symbol, int node);
}
```

Listing 4.5: The `NodeStateConsumer` interface

sibility of storing sequence numbers per node and checking whether any sequence number is a duplicate. The classes implementing the network model are known to the `PDASink` as consumer and to the assertion evaluator as network model.

As can be seen in the `NetworkModel`, the internal state is stored in individual `Attribute` classes. These classes provide chronological access to the attribute's values. There the distinction is made between *possible* values and *verified* ones. If the *verified* method (`getAttributeValue`) returns `null`, then `getPossibleAttributeValue` may be invoked and will return a value if there is one, even if it is not *verified*.
The implementation of `Attribute` holds a list of `AttributeValues` in reverse chronological order, such that the newest one comes first. When the attribute's value is to be looked up for time t_1, this list is traversed to find the latest value supplied earlier or equal to t_1. Assuming that the value was recorded at t_2 (where $t_2 \leq t_1$), this value is the *possible* value for the attribute at time t_1.

To be sure that the value really is valid at t_1, it must be checked that no messages that could have changed the value of the attribute got lost between t_2 and t_1. This is done via the network model, which has the information about missing sequence numbers per node. Another requirement is that there is some time between t_2 and t_1 ($t_1 - t_2 > THRESHOLD$) to compensate for possible errors when synchronizing the messages to a global clock. The $THRESHOLD$ will usually be in the range of a

```
package ma.model;
public interface ConnectivityConsumer {
    /**
     * Report the neigbhors for a node at timestamp
     */
    void reportConnectivity (int node, Set<Integer> neighbors,
          long timestamp);

    /**
     * Report the neigbhors update for a node at timestamp
     */
    void updateConnectivity(int node, Set<Integer> newNeighbors,
           Set<Integer> removedNeighbors, long timestamp);
}
```

Listing 4.6: The `ConnectivityConsumer` interface

few milliseconds. The third requirement for a *verified* attribute value is that the node receives at least one packet from the sensor node after t_1, again without missing packets in-between, to ensure that it is still alive and has neither rebooted nor suddenly stopped sending messages for whatever reason. Figure 4.5 illustrates the conditions for an attribute value to be *verified*.

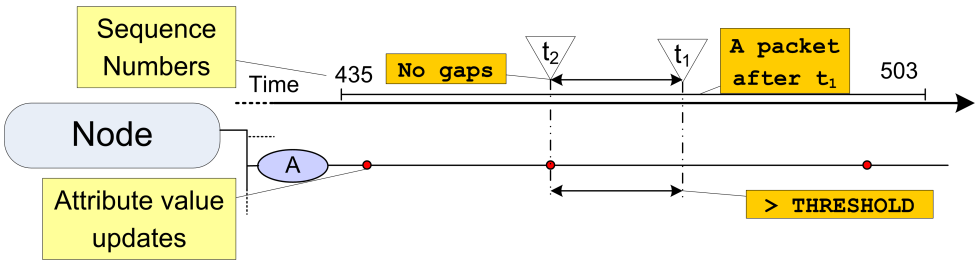

Figure 4.5: Illustration of an attribute's value that is requested for time t_1 and expected value comes from t_2.

4.3.3 Assertion Evaluator

The assertion evaluation consists of a main interface that defines the methods used for triggering an evaluation as well as the event-based notification system for keeping listeners informed about the evaluation results. The interface `AssertionEvaluator` is given in listing 4.7 and implemented in class `BasicAssertionEvaluator`. The `AssertionEvaluator` interface is used by the *packet parser* to kick off assertions that have to be evaluated. The only thing supplied is the ID of the assertion, the time of evaluation, the evaluation constants and - if it is not a global assertion - the *local* node for this evaluation, that is the node, that triggered the evaluation of the assertion.

The assertion evaluator uses the `AssertionResolver` to retrieve the assertion that belongs to the supplied ID. In the available implementation, which can be found in the class `FileAssertionResolver`, the resolver uses the file that is created by the preprocessor and uses the available information to map IDs to assertions.

```
package ma.eval;
public interface AssertionEvaluator {
 /**
  * Evaluate an assertion for a node.
  */
 void evaluate(int assertionID, int node, long time, long delay,
       Map<String, Object> evaluationConstans);

 /**
  * Evaluate a global assertion.
  */
 void evaluate(int assertionID, long time, long delay,
       Map<String, Object> evaluationConstans);

 /**
  * Register and unregister a result consumer.
  */
 void registerResultConsumer(EvaluationResultConsumer consumer);
 void unregisterResultConsumer(EvaluationResultConsumer consumer);
}
```

Listing 4.7: The `AssertionEvaluator` interface

A core part of the assertion evaluator is the parser that is used to transform an assertion into an abstract syntax tree (AST), a tree that reflects the assertion in a form that can be easily evaluated. This section will only contain discussion about technical decisions; for information about the assertion language see section 3.2.5 or for a EBNF appendix B.

The parser is generated using the Java compiler compiler (JavaCC). The generated class is called `AssertionParser` and can be found with all other parser-related classes in the package `ma.parser`. As the parser works internally with data streams, and is therefore somewhat inconvenient to setup just to parse a single assertion, there is the class `StringAssertionParser` that allows easy parsing using a singleton parser that is re-initialized for each assertion in order to avoid side effects. The interface for the `StringAssertionParser` provides a method that takes an assertion in the form of a String and returns an `Assertion` object that encapsulates the AST. Snapshots, as found in the source code of a sensor network application, are a list of at least one attribute name, whose value is updated by the snapshot. These lists can also be parsed by the `StringAssertionParser` that then returns a `Snapshot` object that contains the list of attributes that are updated by the snapshot.
The abstract syntax tree of an assertion is built out of abstract `Expression` objects that allow inspection using the visitor pattern. Whenever a class wants to inspect an `Expression`, it implements the `AssertionTreeVisitor` interface and calls the `accept` method on the `Expression` object it wants to inspect. In doing so it passes itself as an argument, and the actual class that extends `Expression` will call the method corresponding to its type on the visitor interface. The listings 4.8 shows the visitor pattern related part of `Expression` and listing 4.9 gives the visitor interface that contains a method for each of the subclasses of `Expression`. More information about the visitor pattern can be found in [5]. Figure 4.6 shows the class hierarchy for

```
package ma.parser.tree;
public abstract class Expression {
  ...
  public abstract void accept(AssertionTreeVisitor visitor);
  ...
}
```

Listing 4.8: An excerpt from `Expression` the parent class to all assertion AST components

```
package ma.parser.tree.visitor;
public interface AssertionTreeVisitor {
  void visitAssertion(Assertion assertion);
  void visitConstantExpression(ConstantExpression constant);
  void visitIdentifier(Identifier identifier);
  void visitBuiltinFunction(BuiltInFunction function);
  void visitNegation(Negation negation);
  void visitIntegerOperation(IntegerOperation operation);
  void visitBooleanOperation(BooleanOperation operation);
  void visitComparison(Comparison comparison);
  void visitSet(SetExpression set);
  void visitDynamicFunction(DynamicFunction function);
  void visitEvaluationConstant(EvaluationConstant constant);
}
```

Listing 4.9: The `AssertionTreeVisitor` interface

`Expression`.

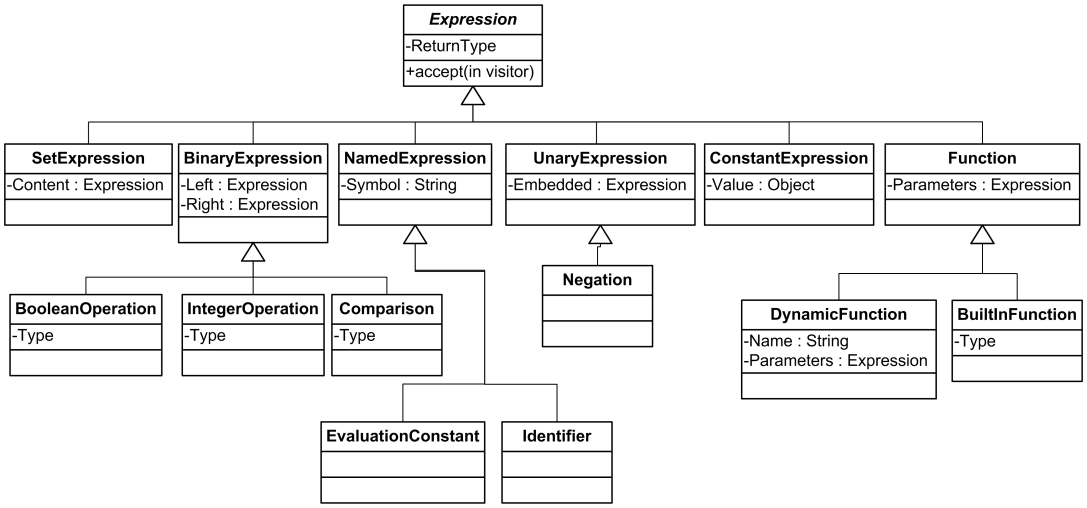

Figure 4.6: The class hierarchy of `Expression`.

There are several implementations of the tree visitor, most notably the `Evaluating-TreeVisitor` that, as its name suggests, evaluates an assertion. To do so it traverses the AST and recursively tries to evaluate subtrees until everything is evaluated up to the root of the tree. If there are any problems with the evaluation process - usually

due to unavailable attribute values - the *runtime exception* `EvaluationException` is thrown, which should be caught be the class that started the evaluation.
Another visitor is the `TypeCheckingTreeVisitor` that checks the types in an assertion. It also recursively tries to derive the type of subtrees until the root is assigned a type. Every element in the tree is checked to ensure that the type of its children are in conformity with it. The root of an assertion, for example, is expected to return a Boolean value. When errors are detected in the tree, an exception with the name `TypeCheckException` - similar to the `EvaluationException` - is thrown that needs to be caught and handled by the class initiating the type check. The other visitors are either for debugging purposes like the `PrintingTreeVisitor` or for extracting certain parts of an assertion, like the evaluation constants found in an assertion, that are then expected to be sent together with the evaluation request.

When parsing an assertion or a snapshot, a wrapper object called `Assertion`, respectively `Snapshot`, is created that holds not only the structure of the parsed text, but also meta information like the file the assertion was extracted from, the line it was found on, the ID that will be received from the nodes and the original assertion text that was passed to the parser. The `StringAssertionParser` will not fill in this information, but when receiving an assertion via the provided `AssertionResolver` this information is supplied.

After an assertion is evaluated for the first time, all listeners that subscribed with the `AssertionEvaluator` are notified that a new assertion evaluation was triggered and receive an `EvaluationResult` that holds all the information about the evaluation. If the assertion could not be evaluated, or contains non-verified values, it is evaluated again after a certain delay and every listener gets an update event, whenever such a reevaluation takes place and part of the result changes.

The `BasicAssertionEvaluator` also takes care of evaluating global assertions. For this it subscribes as an `AttributeChangeListener` to the `NetworkModel` and evaluates global assertions whenever an attribute that belongs to the assertion changes. The global assertions are received via the `AssertionResolver`'s method `getGlobalAssertions` that returns all global assertions. The implementation in the `FileAssertionResolver` loads all global assertions from a file where one assertion is given per line.

To add a user-defined function to the assertion language, a developer has to implement the interface `CustomFunction`. For such a function, the following information must be known: the name, the number and types of the parameters and the return type. Whenever the name is encountered inside an assertion, the method `evaluate` is called with the values of the parameters. The implementation should process the parameters and return a valid result or throw an `EvaluationException`. Listing 4.10 gives the interface implemented by user-defined functions. The new class then needs to be registered with the `FunctionResolver` that returns a `CustomFunction` object for a given function name or `null`, if the function is unknown. The current implementation is a static one, that is, all functions needed at runtime need to be included in the constructor that then inserts them into a map.

There is a graphical viewer for evaluation results. The viewer is connected to the

```
package ma.eval.function;
public interface CustomFunction {
  /**
   * Get the name of the custom function.
   */
  String getFunctionName();

  /**
   * Get the return type of the custom function.
   */
   Type getReturnType();

  /**
   * Get the parameter types (if there are any).
   */
  Type[] getParameterTypes();

  /**
   * Evaluate the function with the given parameters.
   */
  Object evaluate(Object... parameters) throws EvaluationException;
}
```

Listing 4.10: The interface implemented by all custom functions.

evaluation system by implementing the `EvaluationResultConsumer` interface. The GUI presents the results in a table that gives a brief overview of the current situation. The user then has the possibility to select a result, which is displayed in the detail view showing the user all available information for the result. Furthermore the list of the results can be filtered for specific *outcomes* and *status*. Whenever a new assertion is evaluated, it will show up directly in the GUI and will be updated as soon as the result changes to reflect the current evaluation history.

To run the GUI, it is sufficient to create an instance of `AssertionViewer` and make it visible by using its `setVisible` method. The only thing that has to be provided is an assertion evaluator, which delivers the evaluation results to the GUI.

When using the `PESRunner` to start the assertion evaluator, as it is described in the appendix C, the GUI is started automatically. Figure 4.7 shows the user interface in a situation in which a network of three nodes is inspected. The snapshot was taken shortly after the *target* was switched off. The ceding *leader* tries to transfer the leadership to other nodes but fails as no other node detects the *target*. The selected assertion that is shown in the detail form was triggered by the *leader* and should ensure that the triggering node is the only leader in the network. It is not yet fully evaluated, thus the value of the *isLeader* attribute of the *leader* is rated as *possible*.

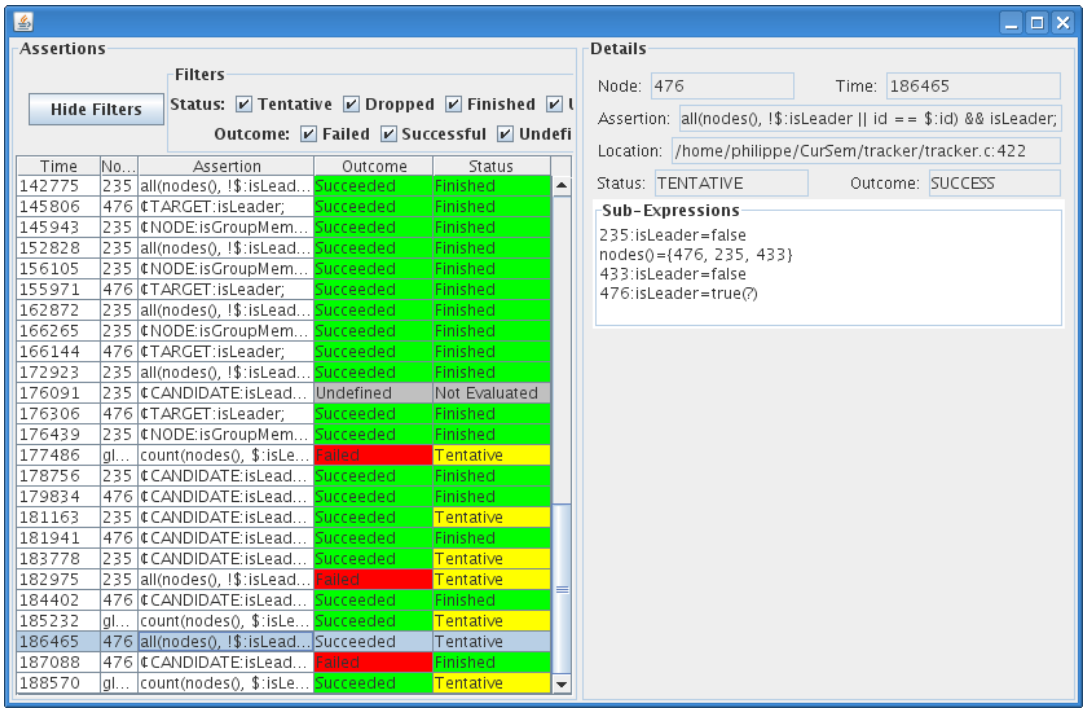

Figure 4.7: The graphical user interface to display. The status and outcome color indicate the current state of the evaluation result.

5 Example Application

In order to be able to test *PES* with a real sensor network, an example application was written to assess the impact that assertions have on the development process and how many additional resources are needed when using *PES*. The evaluation can be found in chapter 6, while this chapter focuses on the application and the assertions used.

5.1 Overview

The goal of the example application is to track the position of a single mobile *target* by computing the average of the positions of the nodes that currently detect the *target*. Therefore the application is called *Tracker*. The idea for the example application is taken from [1] with reduced features for a feasible application.

5.2 Design

The *Tracker* application uses BTnodes, of which every single one has a unique number, called the ID, that is used to address the node and to distinguish between various nodes. Nodes also know their position in terms of x and y coordinates and communicate using their Chipcon radio module. The *target* is just another BTnode which broadcasts, also via radio, beacon messages with a predefined type to make itself detectable.

The *Tracker* nodes form a multihop network in which every node announces itself to its neighbors by periodically sending broadcast messages. The nodes use the announcements they receive to hold a network neighborhood table. To keep the application simple, a greedy geographic routing algorithm is used that always forwards a packet to the neighbor nearerst to the packet's destination, ignoring the possibility of dead ends. Alternatively, an algorithm like *GFG* [2] that deals with these problems using face routing could be used.
Besides point-to-point communication, there is also broadcast communication in which the number of hops a packet is forwarded can be specified by the sender. Every node receiving a broadcast message for the first time that has more hops to take, will re-broadcast it and decrease the *hops-to-live* counter in the message by one. This broadcast communication is used to announce a node to its neighbors and to make a *leader* known to the other nodes.

The basic idea behind *Tracker* is that all nodes that detect the target form a group with a single *leader* node. All nodes in the group send their position to the *leader*, which calculates the position of the *target*. The nodes can hold four different roles

that are described below.

Leader The *leader* is the node that calculates the position of the *target*. There is only one *leader* in the network and every node detecting the *target* knows the *leader*. For most of the time, the *leader* also detects the *target* and is thus also a *group member*, but it is allowed for the *leader* to temporarily not be a *group member* and still act as a *leader* until a new one is found. This could be the case when the target is moving and leaves the detection radius of the current *leader*.

Group member A *group member* is a node that detects the *target* and therefore sends its own position to the *leader* to be included in the calculation of the *target's* position.

Border node A *border node* is one that receives a leader announcement, but does not detect the target itself. This indicates that one of its network neighbors detects the target and is therefore *group member*.

Idle A node has the role *idle* when it does not detect the *target* and has not received a leader announcement.

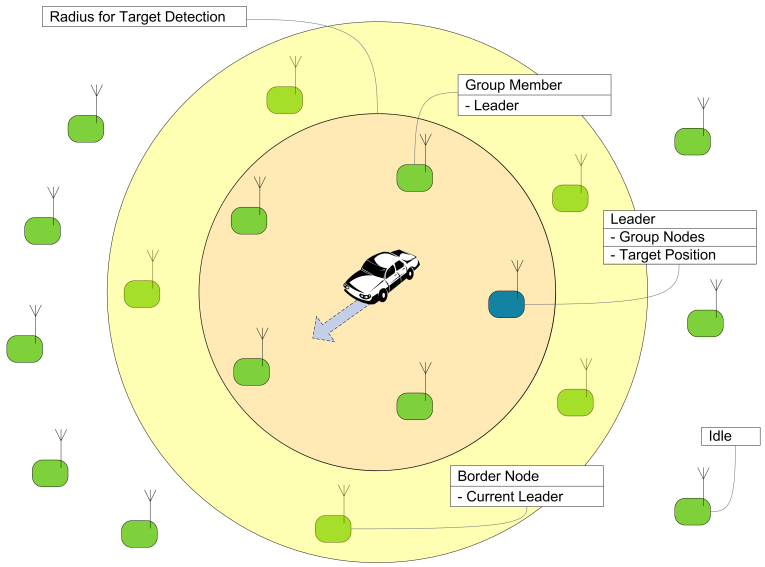

Figure 5.1: Overview of the *Tracker* application, showing the role assignment in a network.

Figure 5.1 shows how the roles are assigned to the nodes and the knowledge that they have about the network. The *leader* knows the position of the *target* and all nodes that detect the *target*, while the other *group member* nodes only know the *leader*. A node that carries the role of *border node* is basically an *idle* node, that has already received a leader announcement message and is therefore aware of the *leader* in the approaching observation group. When the *target* moves into the node's detection range, this node can start reporting to the *leader* immediately.

Figure 5.2 shows the different roles that a node can take and the circumstances under which a node changes its role. It also shows that the only roles that can be held simultaneously are *leader* and *group member*.

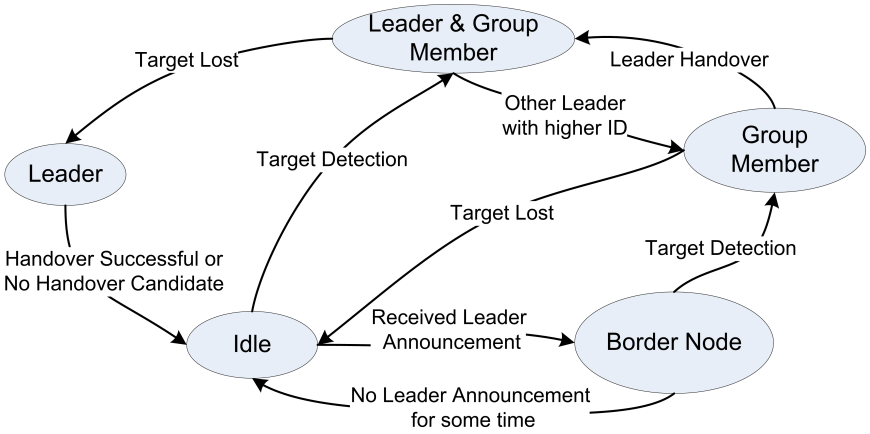

Figure 5.2: The possible role transitions in *Tracker*.

When a node with the role *border node* detects the target, it starts reporting to the *leader*. If no *leader* is known, that is, the node has the role *idle*, it considers becoming *leader* itself. To do so, it first waits for a random time and then, if no other node announces its leadership in the meantime, it announces itself as the *leader*. As the node detects the *target*, it becomes a *group member* in any case. Every node that becomes a *group member* broadcasts the ID of the node it regards as the *leader* to the neighbors to prepare those that might not have detected the *target* at this time for the approaching *target*.
If a node is *leader* and receives a leader announcement of some other node, it stays *leader* if its ID is higher than the ID of the other announcing node or otherwise it takes the other *leader* as its new *leader*. Either way it then announces the current *leader* to its neighbors to notify them about the resolution of the leadership collision. The *leader* does not only calculate the current position of the *target*, but the direction the target is moving in. When the *target* is lost by the *leader*, the *leader* tries to find a new *leader* among the nodes already known to it that lies the furthest in the direction of the *target* movement. If there is a new candidate for leadership, the ceding *leader* sends out a handover request to the candidate, which is confirmed by an acknowledgment. Both nodes then announce the new *leader* and the old *leader* takes the role *idle*.

5.3 Implementation

This section focuses on the implementation details of the application described in the previous section and gives details on the use of distributed assertions.

5.3.1 Communication

Every message used in the tracker application has a common header that incorporates all information needed for routing. The header contains a sender ID, sender position, recipient ID, recipient position and a sequence number. Packet 5.1 shows the basic packet structure.
For sending broadcast messages, the recipient ID is set to `0xFFFF` and the coordinate fields, which in point-to-point messages hold the position of the recipient, are used to count the number of hops the broadcast message still has to take (x coordinate) and the number of hops already taken (y coordinate). The sequence number is used to avoid duplicate packets, especially in the context of broadcasts, where a message could be received multiple times. To distinguish the different messages, the type field of the BMAC message is set according to the message content. Henceforth, this field is simply referred to as the type of message.

Base	0	1	2	3	4	5	6	7
0x00	sender		receiver		sender x		sender y	
0x08	receiver x		receiver y		seq #		... (Payload)	

Packet 5.1: The basic packet structure.

When nodes announce themselves to their network neighbors, the above described basic packet requires no additional payload data. All a node needs to know about its neighbors is the ID and the coordinates that are then used for routing packets. Initially, however, it was planned to allow nodes to provide services to other nodes, like for example, a video camera, whose focus can be adjusted onto the target by sending a service request to the service-providing node. Therefore, there is a one-byte flag field embedded into messages, which allows nodes to announce their services with their identity. The field is not used, but for completeness it is also included in this documentation. Packet 5.2 shows the payload of the announcement packet that has type `0xa0`. Packet 5.3 shows the packet initially intended to invoke services of the node, containing the flags for the service to be invoked, the position of the *target* and the duration of the service, i.e. for how long a camera should film the *target*. Such a service request message would be sent with message type `0xa1`. The node announcement message is sent only to the direct neighbors via broadcast. In case of a node that provides a service, announcements could be sent over more than one hop, depending on the number of nodes that provide services. The service request message on the other hand is sent directly to the node that provides the service.

Base	0
0x00	service (Video|Audio|Unused [6])

Packet 5.2: Announcement packet payload.

To announce the leadership of a node, another broadcast message is used that holds the ID and the position of the leader. This message is only sent to direct neighbors,

Base	0	1	2	3	4	5	6
0x00	service	event x		event y		duration	

Packet 5.3: Service invocation packet payload.

but resent by them, based on the their content and their current state. This means that a leader announcement is forwarded by all nodes that are *group members* and which had a different *leader* before. The leader announcement packet, that is sent with type `0xa3`, is shown in packet 5.4.

Base	0	1	2	3	4	5
0x00	leader ID		leader x		leader y	

Packet 5.4: Leader announcement packet payload.

The other remaining messages are used to request a leadership handover, with type `0xa4`, for sending a handover ACK, with type `0xa5`, and to inform a *leader* about the seen *target*, with type `0xa6`. They all have no payload as the ID of the sender and the sender's position is all the information needed by the recipient.

All the above mentioned packets have a corresponding `struct` in the application code that, having cast the received message payload to the proper `struct`, allows easy access to the different attributes. When accessing the `struct`s, one has to be aware that all packets are in network byte order, and therefore fields with more than one byte need to be converted to host byte order.

5.3.2 Node Attributes

The attributes that are published by the nodes by means of SNAP statements and can therefore be used in assertions are the x and y coordinates as Integers and the flags *isGroupMember* and *isLeader* as Booleans, which indicate whether the node has one of the above mentioned roles.
The x and y coordinates are not measured using a sensor or any other technique, but are stored statically in the application source code. At the start, a node reads its address from EEPROM and, based on the node address, takes the appropriate coordinates from the array `__node_identities`.

5.3.3 Assertions

The core of the tracking application is a thread, which every 10 seconds executes a leader function if the node is a *leader*, or a member function, if the node is a *group member*. In the case of a node being both a *leader* and a *group member*, the leader function supersedes the member function.
As a *group member*, a node sends a target detection message to the *leader* node to inform the *leader* about its position and that it is able to detect the *target*. There the assertion

leader:isLeader

is inserted, which checks that the destination of the message really is the *leader*. Another assertion could be added there to check that every other node that is a *group member* sends target detection messages to the same *leader*. Such an assertion would then look as follows.

all(**nodes**(),!$:**isGroupMember** || $:**leader**==**leader**)

In the code, only the first assertion was used since the second one crossed my mind during the evaluation and was therefore not inserted into the code.

As the *leader*, a node takes all reports it received in the previous interval - the time between two thread cycles - and calculates the *target*'s position as well as the direction of the *target*. The *leader* node has to ensure that every message it includes in the calculation of the *target*'s position really originates from a node that detects the *target*. An assertion to verify this was added to the packet handler for target detection messages and looks as follows, whereas ¢NODE is set to the ID of the sending node.

¢NODE:isGroupMember

This assertion was added to the packet handler for incoming target reports and not to the leader function, although the intention is that every node whose position is used in the target calculation is really a *group member* at that time. This decision was taken because a target detection report from a node that is not a *group member* is considered an error, while a node that is not a *group member* during the time the *leader* calculates the *target* position is an inaccuracy, which is hoped to be small, but was accepted in the design phase. The assertion given above could, of course, still be added to the *leader* thread cycle, to get a picture of the introduced *target* localization error.
Another assertion that is triggered by the *leader* is

all(**nodes**(), !$:**isLeader** || **id** == $:**id**) && **isLeader**;

that can be found at the beginning of the leader function. It asserts that the local node's *isLeader* attribute is set to true and that there are no other nodes with this attribute set to true.

When the *leader* lost the *target*, it tries to hand the leadership over to a node that recently reported the *target*, preferably the one lying the furthest in the direction the *target* moves in. After choosing such a node and sending it a handover request, the *leader* triggers a delayed assertion, to ensure that the leadership is transferred after 100 ms. The assertion in which ¢CANDIDATE is set to the ID of the selected node that should become the leader is evaluated after 100 ms to confirm the successful leadership transfer. The assertion is given below.

¢CANDIDATE:isLeader

This assertion, though, contains the risk that upon receiving the handover request, the CANDIDATE no longer detects the *target* and therefore ignores the request. If this is not regarded as an error, the assertion could be extended by adding the conditions that the CANDIDATE only needs to be *leader* if it still is a *group member*. The above given assertion would then look as follows.

!¢CANDIDATE:isGroupMember || ¢CANDIDATE:isLeader

To ensure that there never is more than one leader, a global assertion was added to the evaluator to test that only one node has its *isLeader* attribute set to true. This assertion looks as follows:

count(**nodes**(), $:**isLeader**) <= 1;

The evaluation system then evaluates this assertion, whenever the *isLeader* attribute of a node changes or a node appears or disappears.

5.4 Discussion

During and after the development of *Tracker*, the application was tested to ensure that it was running correctly. The group formation and leader election work very well with the transmitter power set to a high value. Unfortunately, the position of the *target* was not very accurate due to the fact that the coverage of a radio signal is not circular, but very much dependent on the environment. A target detection mechanism that does not rely on radio would lead to more accurate results here, but also to more work to detect a target.

To test the routing and the formation of the observation group, the radio transceiver power was reduced. Although the routing worked at times, the setup of the network neighbor table was not very stable, nodes kept losing their neighbors and routed messages to neighbors that did not receive them. Also the collection of snapshots via SNIF did not work very well, as the *PES* messages were sent with the same transmitter power and the sniffer nodes seemed to miss some of them, which then led to *dropped* assertions because of missed snapshots.

When the radio transceiver power for the *target* was set to a minimum, the *Tracker* nodes detected the *target* quite haphazardly. Not the nodes nearest to the *target* detected it, but just any node. Additionally the nodes tended to detect and immediately lose the target again, which led to many leadership handovers that were often unsuccessful because the selected new *leader* had frequently lost the *target* as well.

The benefit of *PES* was considerable, as it allowed the state of the nodes to be quickly ascertained. Especially when tuning the *leader* election mechanism, *PES* helped a lot with monitoring the leadership in the network or with verifying, for example, that every node in the network assumed a single node to be *leader*. It helped, for example, to detect an error in a case where all nodes pointed to the same node as *leader*, but this specific node was neither leader nor detecting the target.
The rest of the assertions, especially the simple ones which simply check one attribute on another node, were mostly successful in evaluation and expressive when they failed once. The more complex assertions, especially those that involved the whole set of nodes by using the *nodes()* function, had a higher probability of being *dropped*, that is remain unevaluated, because of missing *verified* attribute values due to message loss.

6 Evaluation

The impact of *PES* on an application should be as small as possible. This not only applies to the time a developer needs to integrate assertions and snapshots in his application, but also, to an even greater extent, to the overhead on the nodes themselves.
The goal of this section is to analyze the overhead of *PES* in comparison to its benefits.

6.1 Parameters and Metrics

PES provides some parameters that allow the application's behavior to be influenced. The parameters are given below:

Enable PDA/SNAP This parameter enables *PES* to be switched on and off during runtime. Of course, this is no parameter for subtle adjustments of the impact that *PES* has on an application, but it allows it to be kept ready for when it is needed and minimizes its performance impact when it is not needed. Even with PDAs disabled, there is some CPU consumption for snapshots as the internal variable cache has to be kept up-to-date to be able to send periodic updates as soon as *PES* is enabled again.

Send Hood The user can decide whether he wishes to collect the neighborhood of the nodes via *PES*. Disabling *hood* sending will seldom lead to less messages, but it will decrease the number of bytes sent per message, as an average snapshot does not need to transfer the network neighbors but only contains the attribute values of the node.
In this context there is another configuration possibility. When transferring the *hood*, *PES* switches between sending the full neighborhood (hood send) and incremental hood changes (hood update), in order to decrease the average message size. How many incremental updates are sent until a complete *hood* is sent can be adjusted. When setting this trade-off, one has to bear in mind that using too many updates before a full *hood* is sent might lead to many *dropped* assertions, as a single lost message renders assertions that access the neighborhood not evaluable until a full *hood* is sent.

Periodic Update Interval The time between two periodic updates can be set by a user. This setting has a major influence on the number of sent messages because after every such interval the node publishes its full state, that is, it broadcasts a snapshot that contains a value for every node attribute. The periodic update snapshot is only omitted if *PES* causes other messages to be sent in order to reduce the load imposed on the network.

Assertion locations The place where assertions are located in the code is important for the number of message that are sent in addition to regular communication when *PES* is used. An assertion that, for example, is added to a periodically called function will be evaluated frequently, which leads to an increase in the number of messages broadcast by *PES*. At the other extreme, assertions could be placed in the code such that an assertion is evaluated only when state changes take place, for example, when a node changes its role. This would decrease the number of messages sent on behalf of *PES* but at the same time reduce the benefit of *PES*, as it decreases the amount of code monitored by *PES*.

There are several basic metrics that can be used when a sensor network application is to be evaluated in order to compare it with other implementations. The following list gives possible metrics and some suggestions on how these metrics can be influenced using the existing parameters in *PES*.

Energy As one of the limited resources, it would be interesting to measure the increased power consumption when comparing an application with and without *PES*. Unfortunately it is labor-intensive to directly measure the power consumption with a high level of accuracy. As an alternative, it is possible to count the node operations whose power consumption is known or that are known to strongly influence the battery drain, like radio communication that can easily be counted in terms of the number of messages and the number of bytes sent. Other candidates for increased energy consumption are additional wakeups from a low-power state and increased CPU usage due to snapshot and assertion processing. These two, however, are not considered by this study as they, again, are hard to estimate and it is assumed that they play a secondary role in the energy consumption imposed by *PES*.

Memory Another critical resource in many sensor nodes is the available memory. When using *PES* the memory usage is of course increased, but it does not seem to be critical - at least in conjunction with BTnodes, which are equipped with a relatively large memory when compared with other nodes.

Computational overhead Although sensor nodes are generally limited in terms of processor speed, this is no issue for *PES* as it does not perform CPU-intensive tasks on the nodes.

The accuracy of *PES* is not evaluated, as it is assumed that *PES* always delivers correct results based upon the assumption that the clock synchronization error is set to an appropriate value. The benefit, however, that a user yields from making use of *PES*, can be measured by counting the *dropped* assertions. These are the assertions that could not be evaluated because a snapshot message got lost or because the assertion's evaluation time lies too close to the change of a node attribute value that is relevant for the assertion evaluation. An obvious goal is therefore to keep the number of *dropped* assertions as small as possible.

As sensor nodes have a limited energy budget and this resource is assumed to be influenced most by *PES*, the attention is focused on minimizing the power consumption, which correlates rather strongly with the number of messages sent and their size. To

evaluate *PES* only the parameter *periodic update interval* (PUI) was varied, as it is assumed to have the biggest influence on the number of messages sent and thus the most influence on the energy consumed by a node. To rate the difference between PUI settings, the following metrics were collected:

# of messages The number of messages sent was counted separately for *PES* and *Tracker*. In order to estimate the packet loss, the received *PES* message were additionally counted on the backend system.

# of bytes transferred The size of the different messages was summed, again once for *PES* and once for *Tracker*.

# of evaluated assertions The number of evaluated assertions was split into two groups: Local assertions and global assertions. A local assertion is invoked by a PDA statement in the source code of *Tracker*. The number of global assertions denotes the number of times that the single global assertion, which checks that only a single *leader* is present, is evaluated. Such an evaluation takes place whenever the attribute *isLeader* changed on a node.

# of dropped assertions The number of dropped assertions, which, again, is counted for global and local assertions separately, states how many of the above mentioned assertion evaluations were *dropped*.

6.2 Evaluation Setup

The main goal of this evaluation is to estimate the amount of additional radio communication introduced by the use of *PES* in *Tracker*. To be able to measure the number and the size of messages, the *PES* connector and the example application were equipped with counters, which are inserted into their code to allow tracking of the number of messages, and thus facilitate comparison of the communication overhead by attributing each message to either the *Tracker* or *PES*. The aim is to draw a graph that shows the number of messages and bytes and how these two values rise during an evaluation run that takes 300 seconds. To facilitate the collection of the values, a thread was added to the node which, once started, first resets the counters for messages and bytes and then periodically prints the values to the terminal.

There were three different PUI settings chosen for evaluation: A periodic update time of 2, 10 and 30 seconds. Initially, a periodic interval of 1 second was envisaged but problems were encountered since the SNIF nodes weren't able to transfer the accumulating messages reliably to the laptop, which was acting as central evaluation instance at that time. Every test setup was run five times and the average values obtained were considered. The power of the radio transceiver was set to such value that every *Tracker* node is able to communicate directly with every other node. This single-hop setup was chosen to keep the order of events similar between the runs with the same PUI. Also, the radio for the *target* was set to a high value to allow all nodes to detect the target simultaneously.

An evaluation starts in an idle state, that is to say, when no *target* is visible. All nodes are initially waiting for it to appear and only exchange message to keep the neighbor

tables updated. At second 120 the target becomes visible to all nodes, which triggers them to select a *leader* and all those nodes that detect the *target* start sending regular updates to the *leader*. After 240 seconds the *target* disappears and the former *leader* first tries to transfer the leadership to another node but fails, as no node sees the target anymore. The test ends at time 300 seconds.

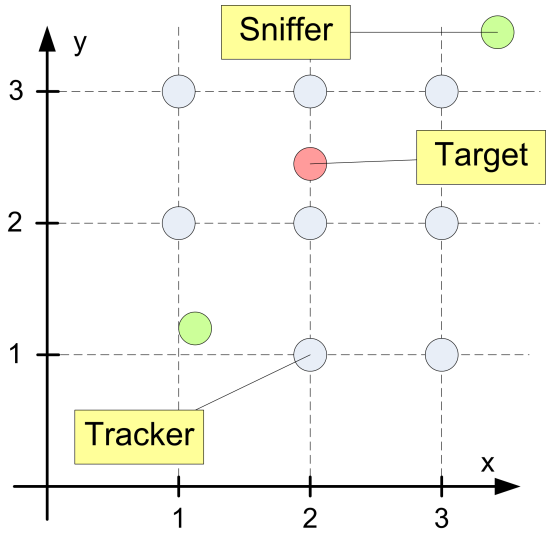

Figure 6.1: The setup used for the evaluation.

As depicted in figure 6.1 the network consisted of eight nodes which are placed in a grid layout and assigned x and y coordinates from 1 to 3, while position 1,1 was left out due to a lack of USB ports. Two SNIF nodes were used to collect the *PES*'s messages. The target was located at a central point and did not move during the test.

6.3 Results

The collected results are given in the figures 6.2, 6.3 and 6.4 that show a timeline displaying the entire test run time from second 0 to second 300. The accumulated number of bytes and messages that were sent per node on average are plotted. The values were calculated as follows: For every evaluation run, the average value per node was calculated and the five resulting values (one for each evaluation run) were averaged again, thus arriving at the average values given in the graphs. Also depicted is the standard deviation between the five evaluation runs. The results are listed in increasing periodic update interval (PUI) order.

It is immediately apparent from figure 6.2 that in the first setting, with a periodic update interval of 2 seconds, the network traffic is dominated by the periodic updates that - together with the other *PES* messages - make up roughly 150 of the 200 messages sent in total by a node on average. The moment the *target* is activated is not visible in the graph when looking at the sent messages. The few additionally sent messages during the time the *target* is activated disappear among the large number of periodic updates. The same holds true for the transferred bytes, so that this plot also

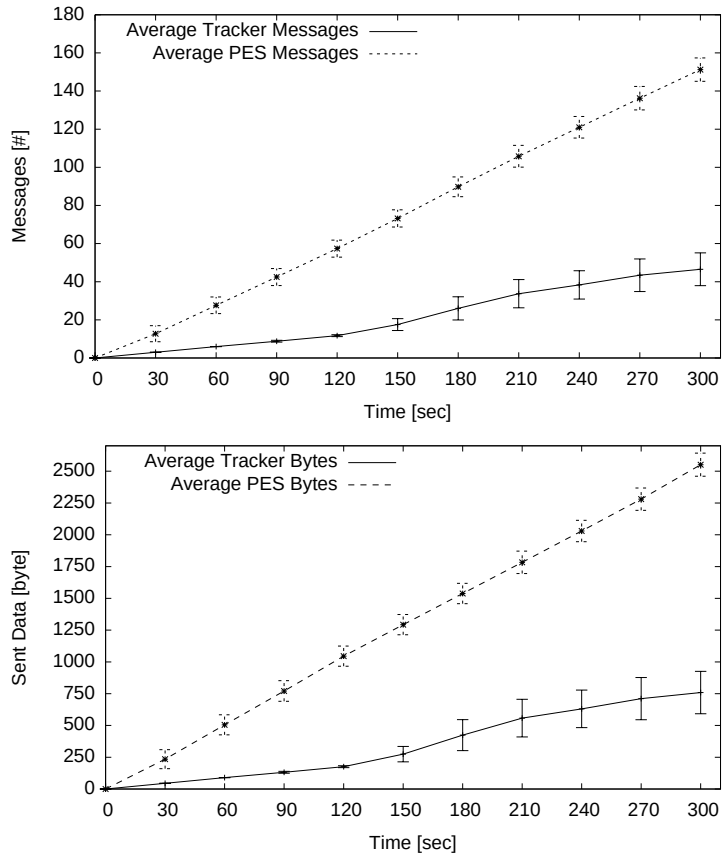

Figure 6.2: Timeline with the number of bytes and messages sent for a periodic update time of 2 second.

forms a straight line with a slope of $8\frac{1}{3}$ bytes per second. The target that appears is only reflected in the data transferred on behalf of the *Tracker* application, for which the *leader* starts announcing its leadership and the *group member*s start sending periodical messages to the *leader* in order to inform it about the detection of the *target*.

For a periodic update interval of 10 seconds, shown in figure 6.3, the appearance of the target at time 120 sec becomes more visible, especially when looking at the standard deviation that raises from $\frac{1}{5}$ to $\frac{1}{2}$ for the number of *PES* messages. This increase in deviation can be attributed to the random wait time, which is included in the leader election mechanism and the fact that the target is started manually and therefore minor time delays are expected when turning the target on. The messages sent by the *Tracker* code should stay roughly the same as there is no change in its parameters, and only the scale in the plot changed such that the increase in messages becomes more apparent. The only possible change to *Tracker* - as compared with a periodic interval of 2 seconds - could be that there is less communication on behalf of *PES* that would then lead to less collisions with *Tracker* messages. The assumption is that this reduced number of collisions could lead to fewer messages sent overall because a leader announcement is received by all nodes. Therefore, it is less likely that another node simultaneously becomes *leader* and, consequently, there is no need

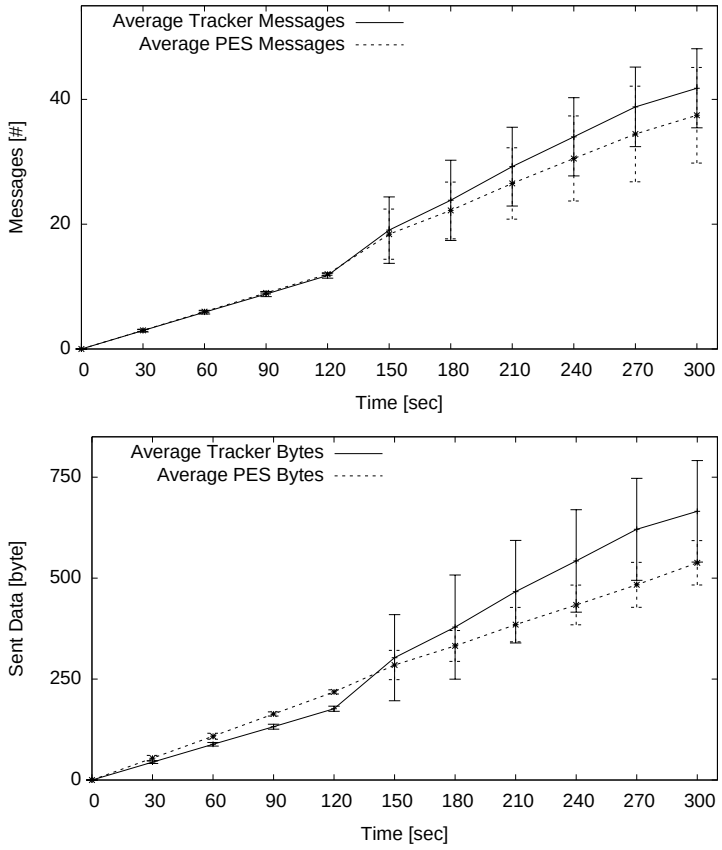

Figure 6.3: Timeline with the number of bytes and messages sent for a periodic update time of 10 seconds.

for leadership resolution that causes more messages to be sent by *Tracker*.
It can clearly be seen that the number of messages rises for *PES* when the target appears while the number of bytes sent per second stays nearly constant. This is due to assertions triggered in the code that lead to multiple messages. Simultaneously the periodic update is omitted. As a periodic update message is rather large, the multiple messages triggered by the appearance of the *target* equate roughly to the same as a single periodic message. In sum, *PES* nearly doubles the number of messages for *Tracker* when a periodic update interval of 10 seconds is used.

The influence of *PES* drops, as expected, even more when setting the periodic update interval to 30 seconds as can be seen in figure 6.4. The number of messages that are directly caused by assertions in the nodes' role function can be seen between the time 120 and 240 seconds, when the number of messages broadcast by *PES* in a 30 second interval is nearly doubled. On the other hand, it shows that *Tracker* still has potential for optimization, as about half the traffic is caused by the announcements sent to network neighbors, which is not only done during active times, when a *target* is visible, but also during idle times.

Figure 6.5 shows the number of messages sent by an average node and received by

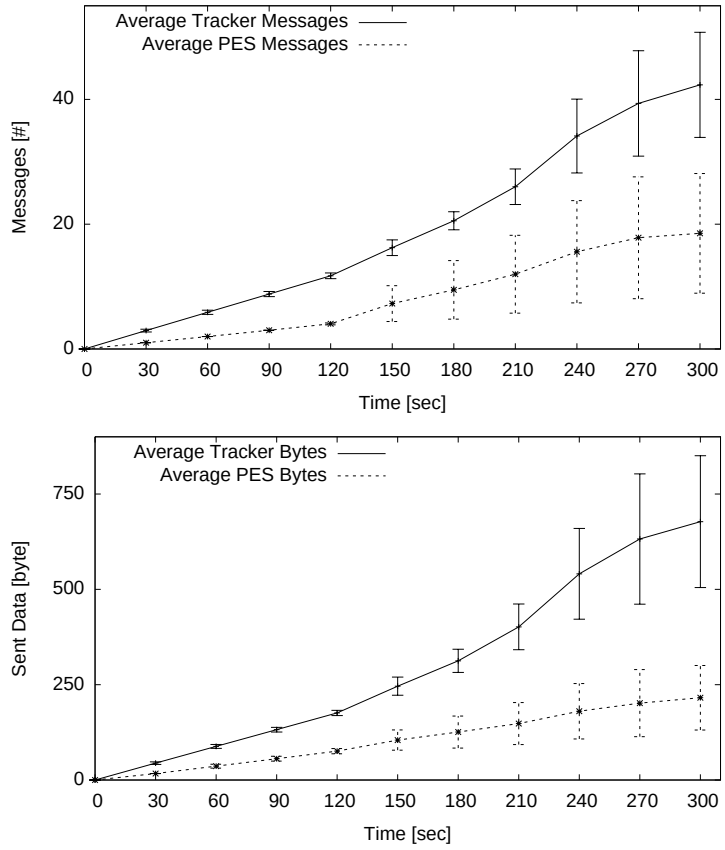

Figure 6.4: Timeline with the number of bytes and messages sent for a periodic update time of 30 seconds.

the evaluation system during the 300 second evaluation run. It is clearly visible that the number of messages sent by *PES* is dominated by the periodic updates for short periodic update intervals. It can also be seen that the message loss for the shortest PUI is much higher than for the other two PUI values. The percentage of lost *PES* message for the three PUI settings of 2, 10 and 30 seconds is 9.83%, 1.80% and 2.42% respectively, equating to 27.08, 1.33 and 0.88 messages on average.
At first sight, the variation in sent *Tracker* messages seems strange, after all, there was no change in the parameters of *Tracker*. This variation, that is, the slightly higher number of messages that were sent with a PUI of 2 seconds, results from the congestion in the network that hindered the leader election to run smoothly. I conjecture that a node missed a leader announcement from another node and then - as it also detected the *target* - announced itself as a leader. The following negotiation to find a single leader is, most likely, the reason for the increased number of *Tracker* messages.

Figure 6.6 shows the assertions that were evaluated on the central evaluator. It includes the total number of assertions, the global assertions and the *dropped* assertions. All global ones were *dropped* due to a design problem described in section 3.4.3. Therefore, the figure also contains a visualization of the number of the dropped local assertions, that is, the dropped assertions whose evaluation was triggered directly by

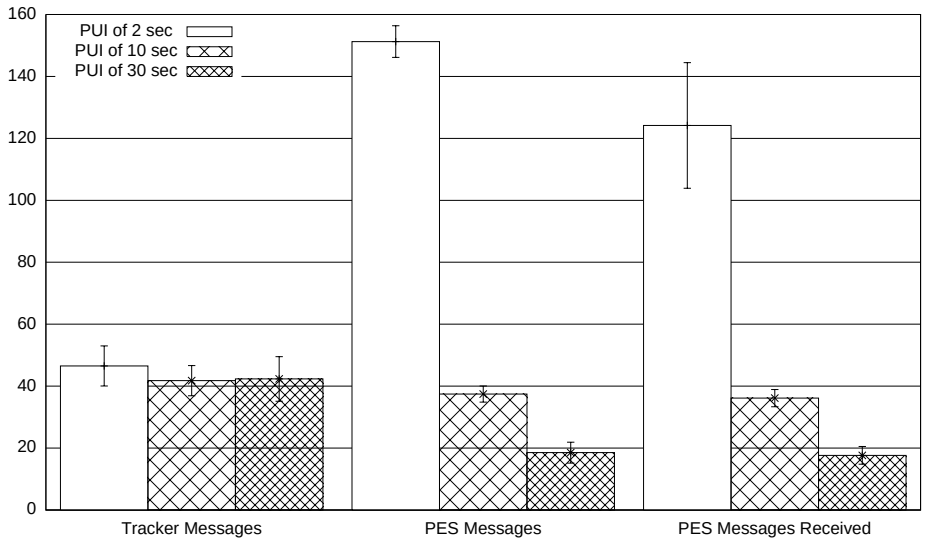

Figure 6.5: The average number and the standard deviation of messages sent on behalf of *Tracker* and of *PES* and the received *PES* message on the assertion evaluator per node on average.

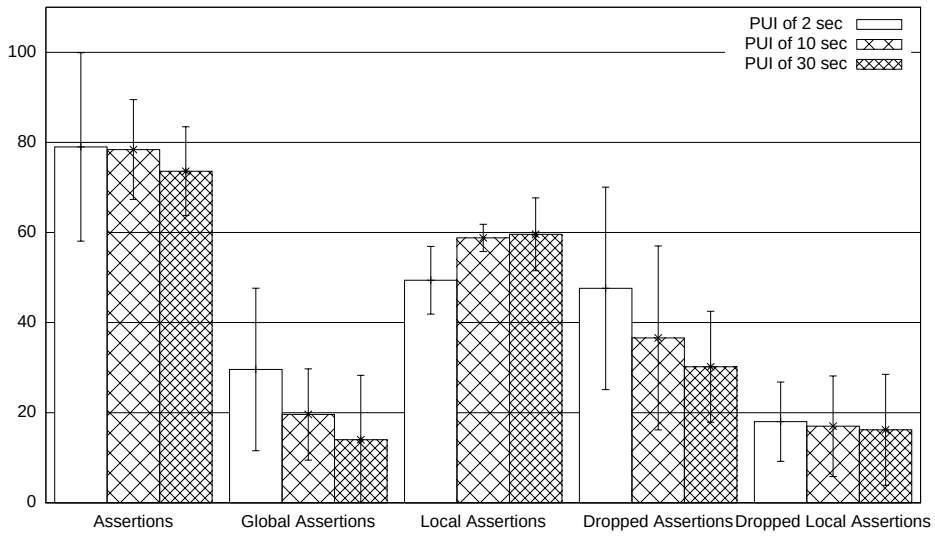

Figure 6.6: The average number and standard deviation of local and global assertions evaluated after 300 seconds in total.

a node.
The difference in evaluated global assertions between a PUI of 2 and 10 is, surprisingly, bigger than the difference in evaluated assertions in total. This means that more assertions are triggered by PDA statements in the code for a PUI of 10 seconds than a PUI of 2 seconds. Whether this is due to lost assertions triggering messages, or due to an non-functional *Tracker* application that skipped the node duty cycle

and thus did not invoke the assertion in the first place is hard to tell. In any case, as already mentioned above, nearly 10 percent of the *PES* messages got lost with a periodic update interval of 2 seconds.

Another interesting aspect is the number of global assertions that is inversely proportional to the PUI. Because global assertions are evaluated whenever an attribute in the model changes, this raise in global assertion evaluation is the effect of many changing attributes, which supports the thesis stated above that the nodes have difficulties selecting a leader. Whenever a node becomes leader or stops being leader it changes its *isLeader* attribute or a node that receives a new leader changes its *leader* attribute. As soon as the network model receives these changes, the global assertion to check that at most one node is leader, is evaluated by the assertion evaluator.

With increasing PUI, even when only local assertions are regarded, the absolute number of *dropped* assertions decreases, even though there are more local assertions evaluated. This suggests that the PUI interval should be at least as high as 30 seconds, as most metrics are best for this PUI value: There is the smallest number of dropped assertions and the least number of messages and bytes sent per node. Given this result, it is probably advisable to try PUI values larger than 30 seconds, to see whether the performance drops, as I initially thought it would with a PUI of 30 seconds.

When comparing the number of messages and bytes per node to the *Tracker* application, the results have to be taken with care as *Tracker* is not optimized to use as few messages as possible and could certainly be tuned to work with fewer resources at a similar performance level. Another thing to keep in mind is that the form of the assertion will also have an influence on the number of of *dropped* assertions. An assertion that uses many different nodes is more likely to be dropped since the probability for this is equal to the probability that any of the nodes used in the assertion has lost a snapshot and its attribute values are therefore only *possible*.

7 Conclusion

The PDA evaluation system designed and implemented in this thesis, improves the visible state of wireless sensor network applications and allows a developer to use distributed assertions to identify unexpected and probably unwanted behavior. This chapter revises the aims given in section 1.2, discusses the limitations imposed by the design and lists possible future work.

7.1 Contributions

The assertion language presented in section 3.2.5 was designed to allow a developer to describe the assumptions made on the state of the network in the form of distributed assertions. The basic components available in the language were chosen after inspecting different WSN applications and proved to be sufficient for a representative example application. With custom functions, a developer is able to adapt the language to his needs. Still, since many necessary assertions arise when an application is actually implemented, further extension may be required in the future.

Sections 3.4.2 and 3.4.3 describe the measures taken to allow a reliable evaluation of distributed assertion. These sections explain how the values of the internal node state can be collected and put into a global picture of the network state. Furthermore, they elaborate on the conditions that have to be fulfilled in order for an attribute value to be regarded as *verified*, in the presence of message loss and clock synchronization inaccuracy.

PES implements the defined assertion language and evaluation algorithm in order to guarantee a correct evaluation of assertions whenever possible. Integrating SNIF, *PES* provides everything needed to include PDAs in an application, to collect the state of the nodes and to decide which assertions have to be evaluated. The evaluation results are then presented to the developer, to give him insight into the state of the network and allow him to identify possible problems and their causes. As the connection to the nodes is realized via radio communication, *PES* can be used during early development in the laboratory as well as after the deployment of the application in the field.

7.2 Limitations

A limitation of the current design is that it requires all assertions to be placed within the code. While this is a key restriction for a passive system, for which nodes only publish their state, it might prove a severe drawback when an application is deployed and it turns out that the already inserted assertions are not sufficient to track down

an error. It is therefore required that the assertions and the location they are placed at are well-chosen in advance.
A restriction imposed by the current version of *PES* is the inability to evaluate global assertions satisfactorily. A solution for this problem has already been presented in section 3.4.3. The implementation of the proposed solution, though, requires a re-design of the interface between the assertion evaluator and the network model and has not been implemented for the current version of *PES*.
An attribute value that is received from a node for time t, is only regarded as *verified* after/before time $t \pm \Delta$, where Δ is the maximum clock synchronization error, as it cannot accurately be found out at which time the value changed on the node. If an attribute value now changes faster than 2Δ, the system will always regard it as being a non- *verified* value. This might reduce the usefulness of assertions in certain applications or certain parts of an application that have such fast changing attributes.

7.3 Future Work

As already indicated in section 6.3, the reasons for the rather high percentage of dropped assertions should be analyzed in more detail. A better understanding of these reasons should enable this number to be decreased by, for example, enhancing periodic updates with a timestamp that states how long a supplied value has already held. This would reduce the time span during which the network model cannot obtain *verified* attribute values because of lost snapshot messages. On the other hand, a timestamp per attribute would greatly increase the size of periodic updates. Another topic, that could be looked into is how larger periodic update intervals affect the number of dropped assertions. The evaluation showed that this number was - contrary to the expectations - lowest for the largest interval. As larger intervals reduce the resource consumption of *PES*, the goal should be to increase this interval as much as possible.

A mapping between node attributes and global variables in the node application code which would allow the compiler to automatically insert snaphots in the code whenever a variable corresponding to an attribute changes would relieve the developer of the task of manually inserting snapshot statements into the source code. This goal, however, is not easily achievable when attributes are not stored as simple variables, but held within more complex structures, such as a table of network neighbors.

When a snapshot message is lost, all attributes are marked as possibly changed for the time between the previous and the next update. An analysis of the node application could allow *PES* to deduce the type of the lost message from the previously received messages and thus offer the possibility of marking only those attributes as possibly changed which really could have been changed by the lost message. Such a lost message prediction requires control flow analysis as found in today's compilers and might not be feasible in C applications, which allow a direct modification of the control flow, without restricting language features for the sensor node application.

Bibliography

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *icdcs*, 00:582–589, 2004.

[2] Prosenjit Bose, Pat Morin, Ivan Stojmenović, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *DIALM '99: Proceedings of the 3rd international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 48–55, New York, NY, USA, 1999. ACM.

[3] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11, 1998.

[4] Michael De Rosa, Seth Copen Goldstein, Peter Lee, Jason D. Campbell, Padmanabhan Pillai, and Todd C. Mowry. Distributed watchpoints: Debugging large multi-robot systems. In *Proceedings of the IEEE International Conference on Robotics and Automation ICRA '07*, April 2007.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] M. Lodder, G. Halkes, , and K. Langendoen. A global-state perspective on sensor network debugging. In *In Proceedings of the 5th Workshop on Hot Topics in Embedded Networked Sensors*, June 2008.

[7] Matthias Ringwald, Kay Römer, and Andrea Vitaletti. Snif: Sensor network inspection framework. Technical Report 535, Department of Computer Science, ETH Zurich, October 2006.

[8] Kay Römer and Matthias Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *Proceedings of the 3rd ACM Workshop on Real-World Wireless Sensor Networks (REALWSN'08)*, Glasgow, United Kingdom, April 2008.

[9] Arsalan Tavakoli, David Culler, Philip Levis, and Scott Shenker. The case for predicate-oriented debugging of sensornets. In *In Proceedings of the 5th Workshop on Hot Topics in Embedded Networked Sensors*, June 2008.

[10] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.

Assertions Extended BNF

B

```
ASSERTION ::= EXPRESSION
EXPRESSION ::= "!" EXPRESSION
       | LITERAL
       | FUNCTION
       | NAMED_EXPRESSION
       | LOGICAL_EXPRESSION
       | NUMERIC_EXPRESSION
       | "(" EXPRESSION ")"
LOGICAL_EXPRESSION ::= EXPRESSION "==" EXPRESSION
       | EXPRESSION BOOLOP EXPRESSION
NUMERIC_EXPRESSION ::= EXPRESSION INTOP EXPRESSION
       | EXPRESSION COMPARISON EXPRESSION
NAMED_EXPRESSION ::= [EXPRESSION ":"] IDENTIFIER
       | "¢" IDENTIFIER
       | "id"
       | "$"
FUNCTION ::= BUILTIN_FUNCTION
       | USERDEFINED_FUNCTION
BUILTINFUNCTION ::= HOOD_FUNCTION
       | COUNT_FUNCTION
       | NODES_FUNCTION
       | ALL_QUANTIFIER
       | EXISTS_QUANTIFIER
HOOD_FUNCTION ::= "hood" "(" EXPRESSION ")"
COUNT_FUNCTION ::= "count" "(" EXPRESSION ["," EXPRESSION] ")"
NODES_FUNCTION::= "nodes" "(" ")"
ALL_QUANTIFIER ::= "all" "(" EXPRESSION "," EXPRESSION ")"
EXISTS_QUANTIFIER ::= "exists" "(" EXPRESSION, EXPRESSION ")"
USERDEFINED_FUNCTION ::= IDENTIFIER "(" [ARGUMENT_LIST] ")"
ARGUMENT_LIST = EXPRESSION {"," EXPRESSION}
INTOP ::= "+" | "-" | "*" | "/"
BOOLOP ::= "&&" | "||"
COMPARISON ::= ">" | "<" | "<=" | ">="
IDENTIFIER ::= CHAR {CHAR | DIGIT | "_"}
LITERAL ::= BOOLEAN_LITERAL | INTEGER_LITERAL
BOOLEAN_LITERAL ::= "true" | "false"
INTEGER_LITERAL ::= NONZERO_DIGIT {DIGIT} | DIGIT
DIGIT ::= "0" | NONZERO_DIGIT
NONZERO_DIGIT ::= "1" | ... | "9"
CHAR ::= "a" | ... | "z" | "A" | ... | "Z"
```

C Deployment

This chapter describes which steps have to be taken to include *PES* into a sensor node application and how the application is started to allow *PES* to receive and process messages broadcast by the nodes.

C.1 Prerequisites and Configuration

A sensor network application developer who indends to use *PES* to debug and deploy his application first needs to include the *pes.h* header file into every C file he wants to use assertions with. To make *pes.h* accessible to the compiler, it needs to be copied into the application directory, together with *linked_list.h*, another header file, which *pes.h* requires to be in the same folder.

The other necessary changes to the application code are that after initializing the node hardware, the developer also has to initialize *PES* by calling `pda_init` and supplying it with two function pointers, which allow *PES* to access the network neighborhood as described in section 4.2.4. The other call that has to be done on initialization of the node is to initialize the attributes of the node, which is achieved by placing a snapshot into the initialization routine of the application that publishes an initial value for all attributes.
If the developer intends to use the terminal, he can also call *pda_cmds_register_cmds* that registers the commands listed in table C.1 with the BTNut terminal.

Command	*Description*
`pdas (on|off)`	Switch PDA's on or off.
`hood (on|off)`	Switch hood transfer on or off (i.e., transferring the *hood* with SNAP and PDA messages).
`cache`	Print the attribute cache of the node.
`pupd (<INTERVAL>)`	Set the interval for periodic updates.

Table C.1: The terminal commands that are provided by *PES*.

To be able to connect to the *deployment support network* (DSN) provided by SNIF, the computer that will host the evaluation system needs a Bluetooth adapter that can be accessed using the Java Bluetooth API (*JSR-82*). Table C.2 lists all dependencies and the versions used during development, testing and evaluation that are needed by *PES*.

Dependency	*Used Version*
SNIF	Revision 71
Log4J	1.2.15
Commons Logging	1.1.1
Commons IO	1.4

Table C.2: Dependencies required by *PES*.

C.2 Preprocessor

The preprocessor can be run by executing the class `ma.preprocessing.PDAPreprocessor` that takes up to 6 arguments. The first two are mandatory

Node folder The name of the folder that contains the node application that has to be processed. This folder is copied and the contained *.c* files are modified.

Evaluator folder The name of the folder to which all files containing the extracted assertions and the derived attribute types are written.

Assertion file The name of the file, relative to the evaluator folder, where the assertions are written to. Optional, defaults to *assertions.txt*.

Type file The name of the file that will contain all the derived types for node attributes relative to the evaluator folder. Optional, defaults to *types.txt*.

User type file Path to the file that contains the user-specified types. Optional, the file is not needed when all types can be inferred automatically.

Global assertion file The name of the file that contains all global assertions, one assertion per line. Optional, needed only if global assertions are used.

After the preprocessor runs successfully and the generated application is uploaded to the sensor node, *PES* can be launched to monitor the network.

C.3 Running *PES*

To run *PES*, the class `PESRunner` has to be executed. It allows the names for the files that are used by the system to be set. Its `main` method takes three arguments to specify the assertion file, the type file and - if available - the global assertion file.
When executed, the `main` method creates an instance of `PESRunner`, which effectively starts *PES* when the method `run` is invoked. In the `run` method, the system is set up with the `BasicEvaluator` as assertion evaluator, the `AdaptingNetworkModel` as the network model and the `NetworkModelConnectivityModel` as the connectivity model. After the basic system is set up, `PESRunner` starts the GUI and connects it to the assertion evaluator to receive evaluation results.

Development Setup

D

This chapter gives the setup used for testing and developing *PES*. The backend was running on a laptop (Lenovo ThinkPad T61p) using the Linux operating system with the kernel version 2.6.26. The node applications ran on BTnodes Revision 3.
The software used is given in detail in table D.1 on the next page. The table not only lists the direct dependencies of *PES* but also components that are required by SNIF and were a bit arduous to set up.

Dependency	*Version*	*URL*	*Comments*
BlueZ	3.3.0	`http://bluez.sourceforge.net/`	
Bluecove	2.0.3	`http://code.google.com/p/bluecove/`	Bluecove provides the Java Bluetooth API (JSR-82 `http://jcp.org/en/jsr/detail?id=82`) for different systems. When using it with BlueZ, additionally the Bluecove-GPL module is needed.
JavaCC	4.0	`https://javacc.dev.java.net`	Only needed when the compiler gets changed and needs to be recompiled
SNIF	Rev 71	`http://code.google.com/p/snif/`	
Commons Logging	1.1.1	`http://commons.apache.org/logging/`	
Commons IO	1.4	`http://commons.apache.org/io/`	
Log4J	1.2.15	`http://logging.apache.org/log4j/`	
Sun JDK	1.6	`http://java.sun.com/javase/6/`	
BTnut	*Comments*	`http://btnode.ethz.ch/`	*PES* was developed using the same BTnut snapshot distributed in the WSN course of 2008. It can be found at `http://www.vs.inf.ethz.ch/edu/FS2008/WSN/tut/tut0.html` or in the BTnode CVS tagged as *WSNLab2008_r1*.

Table D.1: Development setup.