Self-stabilizing Sensor Networks for Emergency Management

Stefan Lohs, Reinhardt Karnapke and Jörg Nolte
Distributed Systems/ Operating Systems group
Brandenburg University of Technology
Cottbus, Germany
$\{$*Slohs, Karnapke, Jon*$\}$*@informatik.tu-cottbus.de*

Andreas Lagemann
Nanotron Technologies GmbH
Berlin, Germany
a.lagemann@nanotron.com

***Abstract*—Wireless sensor networks gather data, which is then transmitted in a sense-and-send manner to a central sink. The large number of sensor nodes makes configuring them before deployment impossible, resulting in the need for self configuration. The possibly large area that needs to be covered results in the necessity for multihop communication, and is prone to link failures. Therefore, communication protocols are needed that can guarantee a certain delivery ratio even under hard environmental conditions.**

A large scale emergency scenario represents one of these hard situations. One of the most fundamental problems in large scale emergencies is the coordination of rescue workers. To dispense units in an efficient manner without endangering them, a central command station needs as much information about the current situation as possible. Therefore, the application requirements are basically the same, making the usage of wireless sensor networks in emergency situations a viable task.

In this paper we discuss the advantages of using self-stabilization in such sensor networks and present a self-stabilizing cross-layer medium access control/routing protocol for data gathering scenarios.

***Keywords*-Wireless Sensor Networks, Fault Tolerance, Self-stabilization**

I. Introduction

In emergency situations, especially when they occur on a large scale, one of the most pressing problems is the coordination of first responders and rescue workers. Usually, a central command station should be used, which coordinates all activities in order to avoid searching the same area twice, and also deciding when to pull out the rescue crews, if conditions get too risky. But keeping such a central command station informed of the current situation is not easy. The rescue workers can not depend on existing infrastructure which might have been damaged. Rather, they need to rely only on hardware they brought themselves.

Sensor networks which collect environmental data like e.g. temperature values or the number of nearby rescue workers can help the command center see the big picture and dispense their units accordingly. Lightweight sensor nodes can be dropped by the rescue workers while they move around the emergency site. But the data gathered by the sensor nodes has to be delivered to the command center somehow.

The radio modules attached to low cost sensor nodes usually only have a small transmission range, meaning that most of the nodes deployed during an emergency will not be able to transmit to the command center directly. Rather, they need to transmit their information over multiple hops, using intermediate nodes. Due to the unforeseeable conditions in the emergency area, routing decisions can not be made before deployment. Also, network characteristics change often, making robust, adaptive, on-demand communication protocols necessary.

The design of a robust protocol has to take into account the problems specified above. The common approach is to augment an existing non robust implementation with additional error handling procedures for each prospective fault. This leads to very complex code, that has a high memory consumption at runtime and is therefore prohibitive in the kind of system considered here. A single sensor-node usually features only about 1-4 kB RAM and 16-128 kB flash. Additionally, this approach necessarily leaves out all cases not anticipated during the design phase.

This can be avoided by specifying the fault free case and regarding everything that diverges from this as faulty. Self-stabilizing algorithms are designed in that fashion. They are based on a description of the valid local states and guarantee that, despite changes through transient faults, one of these states will eventually be reached again. By restricting the algorithms to work on local information only (i.e., the state of a node and that of its immediate communication neighbors), their implementation can be kept concise and yet they handle a large fault class, consisting of faults that are notoriously hard to detect.

In this paper we present a self-stabilizing combination of MAC - and routing protocol for data gathering (sense-and-send) scenarios, where all nodes need to deliver their data to a single sink.

The self-stabilizing design enables the protocol to compute a data gathering tree in an ad-hoc sensor network taking the communication pattern into account. After a transient fault occurs, the protocol is able to converge back into a stable state and bring the WSN back to service.

The Performance of the presented protocol depends on the depth of the generated tree. When a transient fault occurs,

it is possible that the subtree of the faulty node has to be restabilized.

This paper is structured as follows: Section II gives a brief introduction of self-stabilizing algorithms and the SelfWISE framework we used. SelfTDMA, our cross-layer protocol, is described in section III and evaluated in section IV. We finish with conclusion and future work in section V.

II. Self-Stabilization

Self-stabilization was first mentioned by Dijkstra in his paper "Self-Stabilizing Systems in Spite of Distributed Control" [1]. It has been designed for a network of processors with a set of registers. Each processor possesses a *local view* of the network. This means it has read and write access to its own registers and read only access to the registers of its neighbors. All values stored in the registers of a processor combined are called the state of the processor. The union of all processor states is called the system state.

To design a self-stabilizing algorithm it is first of all necessary to define a predicate: the set of stable states of the system. A stable state describes an error free state of the whole network. To reach such a stable state, a set of rules of the form $guard \rightarrow assignment$ is given. If the guard predicate is resolved to true the rule is called *enabled*. If a rule is enabled the list of assignments *may* be executed.

The operation of a self-stabilizing algorithm uses stepwise execution. At each step each processor checks for enabled rules. An omnipresent controller, called daemon, knows which rules are enabled and decides which assignments are executed. There are three typical daemons: central, synchronous and distributed. The central daemon only allows the execution of exactly one rule at a time. In contrast, when the synchronous daemon is used, each processor with at least one enabled rule runs the assignment part of exactly one enabled rule. The third daemon, the distributed daemon, is a mixture of both base types. Here, a non-empty set of processors with one or more enabled rules runs the assignment part of one rule.

An algorithm described in this form is self-stabilizing, if it meets the two conditions of *convergence* and *closure*. The convergence condition guarantees that the system reaches a stable state from each arbitrary state in finite time. Closure means that once a stable system state is reached, the system stays in a stable state as long as no error occurs. Rule execution never leads from a stable state to a non-stable one. It is possible that no rules are enabled in a stable state. As a result, self-stabilizing algorithms guarantee an eventually consistent system state.

A. Self-Stabilization in Wireless Sensor Networks

To use the approach of self-stabilization in the world of wireless sensor networks, several transformations have to be done. In the following, a computation device is defined as a sensor node with built in RAM, instead of a processor with registers. The network between the nodes is based on wireless communication, not on distributed memory or direct data register access. Considering this type of communication, the abstraction of the local view needs to be redesigned. Based on the broadcast characteristics of the wireless medium, one execution step is divided into three parts: At first, each node shares its local state with its neighbors. In the second part, each node evaluates the rule guards. Execution of assignments is done in part three.

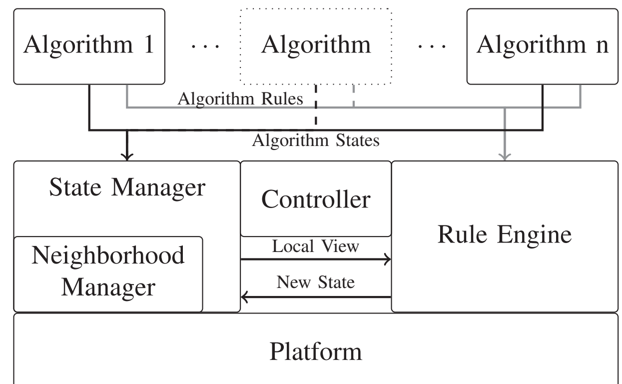

Figure 1. Structure of SelfWISE-Framework

Another problem that needs to be solved is the abstraction of the daemons. In the world of wireless sensor networks, no omnipresent control unit which reaches all nodes exists. There are several transformations for each type of daemon [4], [5], [6]. Considering the standard operation of a system of independent sensor nodes, the abstraction of the distributed daemon seems to be the best match. In the following, the approach presented in [8], [13] is used. Each node decides whether or not to execute the assignment of an enabled rule based on a fixed probability.

If transient faults, e.g. link breaks, occur, they affect the local view of at least one node. The change of the local view can cause a violation of the system predicate, so at least one guard has to be enabled. The enabled rule will be executed and the system starts to converge back to a stable state.

B. SelfWISE

To ease the implementation and execution of self-stabilizing algorithms on wireless sensor networks, Weyer and Turau developed the SelfWISE framework [12]. It is divided into two parts:

The first part provides a language which enables the implementation of algorithms using predicate logic expressions. The second part is a modular middleware which supports the execution of the generated algorithm. Each exchangeable module fulfills a special task. Figure 1 shows a schematic representation of the framework used at each node.

The *Platform* module connects the general parts of the framework with the characteristics of a particular hard-

ware. This module takes care of platform depended tasks like communication and random number generation. For the execution of the algorithms it is necessary to explore and determine a bidirectional communication neighborhood, which is done by the *Neighborhood* module. Based on the explored neighborhood, the *State-Manager* module shares the current state of the node and manages the states of the own neighborhood. The state manager also provides an interface which is used to request the states of neighboring nodes for the evaluation of the guards. The management of rules, evaluation of guards and the execution of the assignment part is done by the *Rule-Engine*. Which enabled rule is executed depends on the implementation of the rule engine. Which module is activated is defined by the *Controller*. The abstraction of the daemon is also included in the *Controller*. If any rules are enabled, it decides whether or not any one of these rules is executed.

The SelfWISE framework is able to handle several dependent and independent self-stabilizing algorithms. Whether or not the composition of algorithms is also self-stabilizing depends on the implementation of these algorithms.

III. SelfTDMA

In this section we describe the SelfTDMA slot assignment scheme. The goal of this approach is to increase the data flow from the data sources to a fixed sink by taking the convergecast communication pattern into account. To optimize the data flow the increasing network load closer to the sink needs to be considered. This network load causes an increasing probability of message collision and congestion. To avoid the problem of collisions we use a time division medium access control mechanism. The slot assignment supports the data flow on paths from data sources to the sink by decreasing latency and avoiding congestion.

The SelfTDMA scheme consists of three succeeding self stabilizing algorithms. Each algorithm has only read access to the results of the previous one, so we can guarantee stabilization if each algorithm stabilizes.

In the following subsections the three self-stabilizing algorithms are presented in detail, followed by the description of the TDMA slot assignment. Section III-A shows the self stabilizing tree construction algorithm. The computed minimal spanning tree is, on one hand, used for routing messages to the sink. On the other hand it is also used to determine shortest paths between leafs and the sink. Section III-B describes the second algorithm. This algorithm counts and shares the number of paths between leafs and sink and the overall count of paths. The computed tree from step one is used as base for the second step. The third algorithm (section III-C) labels each path with a unique number. Based on these numbers, each node is able to determine a set of IDs of paths it participates in. Using these IDs, each node is able to compute the slots it is allowed to send in, and the slots of its children. The slot assignment is described in subsection III-D.

The state of each node, managed by the state-manager, is a composition of the state variables of the three algorithms.

A. Tree Construction

To compute the routing tree the algorithm from Dolev[2] is used. This algorithm generates a minimal spanning tree and provides information about the hop distance to the sink and the ID of the next hop (called parent) at each node. The resulting routing tree is rooted at the sink. This base algorithm of Dolev is expanded to compute a boolean value which is true, if a node has no children.

The state of this algorithm consists of three variables, an ID for the parent, an integer for the hop distance and a boolean for the `isLeaf` flag.

The algorithm is composed of three rules. Rule one, only executed by the root, assigns a distance of zero and an invalid parent ID to its state variables. The second rule selects a parent with minimal hop distance to the root from the set of neighbors. If more than one node with minimal distance exists, the first node of the set is selected as parent. Based on the selected parent, the distance value is set to the parents distance plus one. Rule three determines that a node is a leaf, if no node exists, whose parent register is equal to the node identification of the current node.

B. Exchange of Tree Information

The second self stabilizing part of the SelfTDMA algorithm counts the paths between leafs and sink. The computation starts at the leafs. Rule one sets the amount of paths a leaf is participating in to one. Rule two is applied by all non leaf nodes. Each non leaf node computes the amount of paths it is participating in by adding the number of paths its children participate in. Starting from the leaf, each node on the path to the root computes correct values for the amount of paths in the spanning tree. Finally, the root is able to compute the overall path count of the tree by applying rule three. The result of the computation is propagated back to the leafs using rule four.

The state of this algorithm stores a variable for the amount of paths it is participating in and the overall number of paths in the network.

Please note that the second algorithm stabilizes if and only if the spanning tree algorithm is stable. As long the first step is unstable, the execution of the rules of algorithm two can lead to wrong decisions.

C. Path Allocation

The purpose of the third algorithm is to choose path indices. These indices are stored in a bit array. Each node has to choose an index for each path it participates in. There are three constraints for how each node can select indices: First, no two nodes at the same hop-distance to the root

choose the same indices. Second, the set of chosen paths of one node is a subset of the indices of its parent. Third, the highest used index is equal to the number of paths in the spanning tree.

The algorithm starts at the root. Using rule one, the root node chooses the indices of all paths in the tree, because the root is member of all paths. Rule two is applied by all non-root nodes. It is used to choose an available and free ID from the parent node.

Because of the parallel processing of all nodes it is possible that two nodes that share a parent choose the same ID. This would be a contradiction to the first constraint mentioned before. To dissolve such double selections, each parent node computes a set of chosen paths of its children (rule three). This set is also used to decrease the probability of a wrong selection at rule two. During computation of the already chosen IDs, the parent node is able to detect double selected path IDs. Rule four displays the multiple times chosen path ID and the node ID of one of the responsible nodes. When a child node is causing a double allocation, rule five frees the corresponding path ID.

The parallel execution of all algorithms and changes of neighborhood can lead to the problem that a formerly chosen path ID is not longer available or that a node selects too many paths. Rule six frees IDs which are no longer provided by the parent node. If a node has chosen more path IDs than the amount of paths it participates in, rule seven frees an arbitrary path ID.

The state of the third algorithm consists of two bit arrays representing the IDs of the path the node itself has selected and all IDs its children have selected. Additionally to these arrays two integers show the array index and node ID of a collision detected by this node.

D. Slot Allocation

Based on results of these three self-stabilizing algorithms, each node is able to compute in which slots it is allowed to send and the slots its children are using. The main goal of the SelfTDMA algorithm is to reduce the buffer congestion in the data gathering scenario. To reach this goal the idea of the SPR algorithm [9] is adapted. This adaptation reduces the forwarding latency of packages on the path from leafs to the sink.

Each node is able to compute its slots σ with the following equation:

$$\sigma := \{i \cdot \kappa + (\kappa - (d \bmod \kappa)) | 0 \leq \mathtt{i} < \#\mathtt{p}, \mathtt{P[i]} = \mathtt{true}\}$$

Here, d is the hop distance to the sink, $\#p$ is the amount of paths the node is participating in and P is the bit array of chosen path IDs.

The result of the equation is a TDMA frame divided into subframes for each path from leaf to sink in the spanning tree. First, path one is activated, followed by the subframe for path two and so on. Each path subframe has κ consecutive slots. All nodes of a currently active path with a distance $(\kappa - (d \bmod \kappa)) = 0$ use the first slot . All nodes with $(\kappa - (d \bmod \kappa)) = 1$ use the second slot and so on. After κ hops on the path to the sink the slots are reused. This allocation scheme enables forwarding of a message up to κ slots per path subframe.

Each node allocates one slot for each path it participates in. Thus nodes with higher network load closer to the root have more slots. This decreases the probability of congestion.

The constant κ affects the behavior of the MAC-protocol significantly. A high value of κ increases the amount of hops a package can be forwarded per path activation. It also reduces the probability of collisions caused by the reuse of slots each κ hop on the path. But a high value also causes a bad slot utilization because each subframe contains κ slots. If the length of some paths is lower than κ, the first slots of the subframe will be unused. The length of the whole TDMA frame depends on the amount of paths and the size of κ. As in all TDMA schemes, a long frame increases the latency of messages drastically.

Taking advantage of the minimal spanning tree, κ can be set to three, to avoid collisions within communication distance. If the interference distance is much greater than the communication distance, a higher value of κ should be used.

IV. Evaluation

To evaluate the protocol, we implemented SelfTDMA and SelfWISE for the event driven operating system REFLEX [11]. The evaluation is dived into two parts. First, the protocol is evaluated under controlled conditions in the simulator OMNeT++ [10], an integration of REFLEX was already available [7]. Here, different topologies and different error scenarios are evaluated. The second evaluation is done in a real world setup, based on the eZ430-Chronos platform from Texas Instruments [3].

A. Implementation

The implementation of SelfWISE consists of simple module implementations. The Neighborhood is easily explored by exchanging the local neighbor table. A node that receives a neighborhood message adds the node from which it received the message to its own table. A bidirectional link between node A and B is assumed, if node A receives a neighbor table from B with an entry of A.

The *RoundRobinRuleEngine* evaluates all registered rules at the start of each step. The first enabled rule is marked, and is prepared for execution. The Controller decides, with a user defined probability, if the marked rule is executed.

The state of all algorithms is exchanged periodically. The simple state manager only manages states of neighbors with bidirectional communication links. All states of neighbors connected via unidirectional links are ignored.

The SelfTDMA protocol also controls the execution of the SelfWISE framework. Before the application is started, a setup phase allows the continuous execution of the framework. After the start of the application, the computed TDMA frame is activated. The frame for the path activation is extended by some slots for the execution of the SelfWISE framework. The subframe at the start of the TDMA frame is long enough to run three self stabilizing steps.

B. Simulations using OMNeT++

The goal of the OMNeT++ simulations is to test the behavior of the protocol under controlled conditions. On the used physical layer no errors occur during transmission of messages. Even though in reality the range of interference is larger than the communication range, they are simulated as being of equal size. Instead of error due to interference, different error situations are simulated using a scenario manager. This scenario manager is able to inject link breaks and complete node failures into the network.

Based on this OMNeT++ setup, different topologies were evaluated. The size of the network was varied between 40, 60, 80, 100 and 200 nodes, with an average number of neighbors of 4, 6, 9, 12, 15 and 18 nodes. The evaluated scenarios vary in failure frequencies, failure duration and ratio between link breaks and complete node failures.

The test application was a simple sense-and-send application. Each node generated messages periodically, and transmitted them to the sink. At the sink, the data from the received messages was recorded. With this, the packet delivery ratio and the average latency can be calculated. In the simulation, one node in the network generated a packet and transmitted it to the sink every ten seconds.

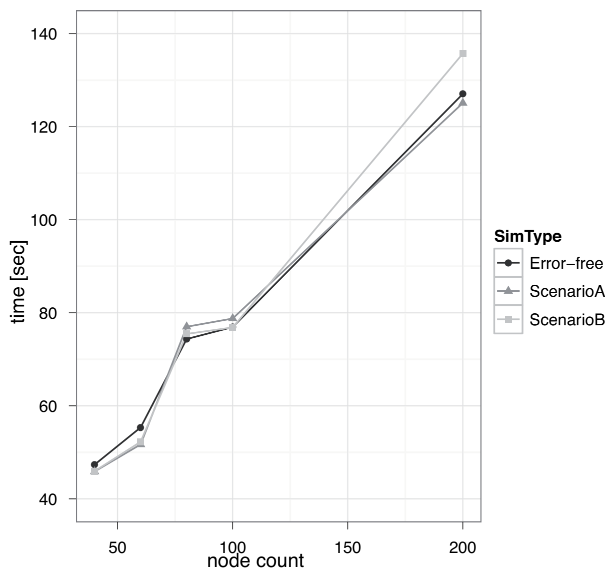

Figure 2. Average Latency in Simulations with average density 15

Figure 2 shows the measured latency values for three different scenarios: In Scenario A, an error occurred every ten seconds, it lasted ten to twenty seconds. Eighty percent of these errors were link breaks. In Scenario B, an error occurred every thirty seconds, it lasted thirty to sixty seconds. Eighty percent of these errors were link breaks. The third scenario contains no errors and is used as reference.

The figure shows that the different error scenarios affect the latency of message transmission only marginal. An increasing node count increases the latency because of the longer paths between leafs and sink.

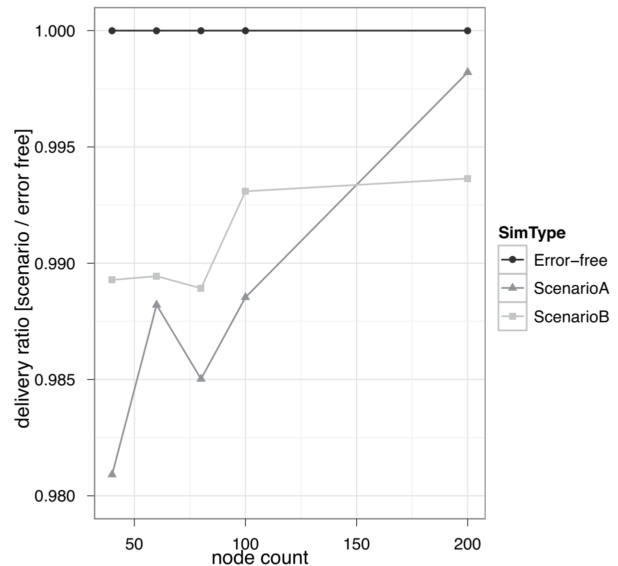

Figure 3. Delivery Ratio for three simulated Scenarios

Figure 3 shows the reached delivery ratio. In the error-free case, the protocol reached a delivery ratio of one hundred percent. Taking a closer look at both other scenarios it can seen that the packet delivery ratio does not decrease significantly. In both scenarios a delivery ratio of 98 to 99 percent is reached.

C. Real world evaluation

In the second part of the evaluation the algorithm was evaluated on modified eZ430-Chronos nodes from Texas Instruments. These sensor nodes have a program memory of 32kB, 4kB of RAM and transmit in the sub-gigahertz band (868MHz). The usage of REFLEX-Code in the simulations and on the real hardware enabled us to keep the necessary software modifications to a minimum. The SelfWISE platform implementation was replaced and extended by a hardware random number generator. For the TDMA, a kind of clock synchronization was necessary. A round synchronization at the start of the application was used, which is of course not as accurate as the synchronization

used in the simulations. In this experimental setup each node, except for the sink, generated a packet every ten seconds and transmitted it to the sink.

The experiment was run with a small three times three grid on two different locations. The first experiment was located at the second floor of the main building of our university. The sensor nodes were placed on the ground and the experiment was repeated three times. Even though no retransmissions were used, an average packet delivery ratio of 67.5 percent was reached.

The second experiment was located at the lawn in front of the main building. The sensor nodes were placed on poles twenty centimeter above the ground. The experiment was repeated five times with the same parameters. The measured packet delivery ratio shows a slightly better result of 70.8 percent, still without retransmissions.

During the experiment, the communication was monitored using passive monitoring. The monitoring tool shows that about ten percent of received packets were invalid. The reasons for this are as yet unknown. One possible explanation for this could be strong interference. Another one would be hardware problems. Other experiments conducted with the same hardware (results submitted to S-cube 2012) seem to show an error in the state machine of the radio module. Which of the possible explanations holds true if any, or if there is a different one entirely, remains to be seen. Experiments are currently underway, which should help in finding the right explanation.

V. Conclusion

In this paper we have presented a robust, self-stabilizing TDMA/routing protocol for wireless sensor networks used in emergency management. Simulations conducted so far show a good stability behavior. When errors occur, the system is returned to a stable state quickly.

Even though the implemented self-stabilizing algorithm does not use any retransmissions, a delivery ratio of more than 98 % was reached in simulations. Using real sensor network hardware from Texas Instruments, a delivery ratio of about 70% was achieved in the real experiments. This represents a huge information gain for central command, and can be further increased by using retransmission and/or aggregation schemes.

Acknowledgment

This work was supported by the Brandenburg Ministry of Science, Research and Culture (MWFK) as part of the International Graduate School at Brandenburg University of Technology (BTU). Current work is funded by the Deutsche Forschungsgemeinschaft in the project ToleranceZone (DFG NO 625/6-1).

References

[1] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11):643–644, November 1974.

[2] Shlomi Dolev. *Self-stabilization*. MIT Press, Cambridge, MA, USA, 2000.

[3] Texas instruments ez430-chronos, http://focus.ti.com/docs/toolsw/folders/print/ez430-chronos.html?dcmp=chronos&hqs=other+ot+chronos.

[4] W. Goddard, S. T. Hedetniemi, D. P. Jacobs, and Pradip K. Srimani. Anonymous daemon conversion in self-stabilizing algorithms by randomization in constant space. In *Proceedings of the 9th international conference on Distributed computing and networking*, ICDCN'08, pages 182–190, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] Maria Gradinariu and Sebastien Tixeuil. Conflict managers for self-stabilization without fairness assumption. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, pages 46–, Washington, DC, USA, 2007. IEEE Computer Society.

[6] Ted Herman. Models of self-stabilization and sensor networks. In Samir Das and Sajal Das, editors, *Distributed Computing - IWDC 2003*, volume 2918 of *Lecture Notes in Computer Science*, pages 836–836. Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-24604-6_20.

[7] Andreas Lagemann and Jörg Nolte. Integration of event-driven embedded operating systems into omnet++ – a case study with reflex. In *2nd International Workshop on OMNeT++*, Rome, Italy, March 2009.

[8] V. Turau and C. Weyer. Fault tolerance in wireless sensor networks through self-stabilisation. *Int. J. Commun. Netw. Distrib. Syst.*, 2:78–98, November 2009.

[9] Volker Turau, Christoph Weyer, and Christian Renner. Efficient slot assignment for the many-to-one routing pattern in sensor networks, 2008.

[10] András Varga. The omnet++ discrete event simulation system. In *Proceedings of the European Simulation Multiconference (ESM'2001)*, Prague, Czech Republic, June 2001.

[11] Karsten Walther and Jörg Nolte. A flexible scheduling framework for deeply embedded systems. In *In Proc. of 4th IEEE International Symposium on Embedded Computing*, 2007.

[12] Christoph Weyer and Volker Turau. Selfwise: A framework for developing self-stabilizing algorithms. In Klaus David and Kurt Geihs, editors, *Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pages 67–78. Springer Berlin Heidelberg, 2009. 10.1007/978-3-540-92666-5_6.

[13] Christoph Weyer, Volker Turau, Andreas Lagemann, and Jörg Nolte. Programming wireless sensor networks in a self-stabilizing style. In *Third International Conference on Sensor Technologies and Applications*, Athens, Greece, 2009.