Mission Statement: ToleranceZone

A Self-Stabilizing Middleware for Wireless Sensor Netzworks

Stefan Lohs, Jörg Nolte
Distributed Systems \ Operating System Group
Brandenburg University of Technology Cottbus
Email: {slohs, jon}@informatik.tu-cottbus.de

Gerry Siegemund, Volker Turau
Institute of Telematics
Hamburg University of Technology
Email: {gerry.siegemund, turau}@tu-harburg.de

***Abstract*—Wireless sensor networks (WSN) can be used in a wide range of monitoring and controlling applications. These networks consist of nodes with sparse resources, which makes application implementation challenging. Therefore, many middleware systems were developed in the last decade.**

Furthermore, unattended and long-living deployments of WSNs need fault-tolerant software architectures. The goal of the TOLERANCEZONE project is to design a self-stabilizing middleware, which supports the development of autonomously recovering and highly fault-tolerant WSN applications.

***Index Terms*—wireless sensor networks, self-stabilization, fault tolerance, ToleranceZone, middleware**

I. INTRODUCTION

Wireless sensor networks (WSN) are suitable for a wide range of applications. Typical examples include, environmental monitoring, automated building control, and intrusion detection.

A WSN consists of small, inexpensive devices, which make large area deployments, in various environments, affordable. The on-chip radio device allows wireless communication between these sensor nodes, making additional infrastructure unnecessary.

Sensor nodes have restricted resources, in particular, small memory and limited processing power. Moreover, wireless communication is prone to frequent link changes, caused by environmental influences, reflections, and interference. An increasing amount of sensor nodes raises the probability of packet collisions. Data loss and bit errors during communication are common.

Consequences are adverse effects on the execution of the application. Eventually resulting in complete node breakdowns, rendering the network useless in the long run.

The typical approach, to deal with such faults, is to enhance an existing implementation with additional error handling procedures for each potential error. Hence, complex code for recognition and handling, needs to be implemented. These routines lead to higher resource consumption, e.g., using up flash memory. In addition, the more complex code is error prone as well. The challenge is to consider all potential fault situations, which is virtually impossible.

In contrast, it is feasible to specify the fault-free states of a system and permanently converge, from each arbitrary state, into this set. This is the idea behind self-stabilizing algorithms. They guarantee, despite changes through transient faults, that one of the stable system states will eventually be reached. No further error handling code needs to be added, hence, the implementation can be kept concise, while being able to handle a large number of faults.

Due to the fact, that sensor nodes are very limited in processing power and memory, algorithms are usually strongly connected to the underlying platform, i.e., hardly portable or reusable. Therefore, middleware platforms for WSN have become quite popular [1], [2].

On the one hand, they abstract from different lower level dependencies, e.g., sensor node platforms or operating systems. On the other hand, they ease the development and the implementation of services, supporting the developer with different interfaces and protocols, thereby, decreasing the probability of errors due the abstraction of lower level constraints.

However, common middleware does not provide mechanisms to avoid, detect, or correct faults during runtime.

Closing this gap is the aim of the TOLERANCEZONE project. We propose to develop a middleware for WSNs based on self-stabilizing mechanisms.

II. SELF-STABILIZATION

Self-stabilization was first mentioned by Dijkstra [3]. He described a distributed network of processors with a set of registers.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. Each processor has a local view of the network, i.e., its own state and the states of its neighbors. A self-stabilizing algorithm defines a set of stable system states and a set of guarded rules to reach such a state. A rule is only allowed to be executed if the guard predicate is resolved to *true*.

The execution of a self-stabilizing algorithm is step-wise. First, all processors check the rule-guards based on the own local view and mark them as *enabled* if true. Second, a non-empty set of enabled rules run their assignment part.

An algorithm described this way is self-stabilizing if it meets the conditions of convergence and closure. The convergence condition guarantees that a stable system state is reached in finite time from each arbitrary state, given that no error occured in the meantime. The closure condition means that the system stays in the set of stable states as long as no error occurs, even though an assignment part of a rule is executed.

To use the concept of self-stabilization in WSNs, the model has to be converted. This transformation concerns the state exchange and the execution of the management. An overview of the preliminary work is given in [4], [5], and [6].

SelfWISE-Framework

C. Weyer et al. developed the SelfWISE framework to ease the implementation and execution of self-stabilizing algorithms in WSNs [7]. The framework consists of two parts:

First, it provides a language and compiler which enables the developer to implement self-stabilizing algorithms in a predicate-logic expressional fashion. The second part provides a modular runtime environment, which supports the generated algorithm.

The state manager module coordinates and distributes a node's current state and the states of all its neighbors. To compute the local view of each node, the neighborhood manager determines a set of bidirectional connected nodes as neighbors. The rule engine and the controller manage and evaluate guards and run the assignment parts of rules.

The SelfWISE-framework was evaluated with several self-stabilizing algorithms. Results of the tree construction, maximum independent set, and vertex coloring algorithm are presented in [8]. The outcome shows, that the measured convergence time of the algorithms, to move from an arbitrary state to a stable state, is better than the calculated worst case. The SelfTDMA protocol, presented in [9], is a complex example of a self-stabilizing algorithm running on the SelfWISE framework. Results of OMNeT++ simulations, involving different fault scenarios and an evaluation with a real hardware platform, are stated in the paper.

These findings show that the SelfWISE framework supports the development of self-stabilization algorithms.

III. ToleranceZone Middleware

The goal of the ToleranceZone project is to aid developers of WSNs to build fault-tolerant systems, with essential operations based on self-stabilizing algorithms. The four idioms which lay foundation of this project are: *shared variables*, *data aggregation and reduction*, *group formation*, and *neighborhood management*.

Architecture

Figure 1 depicts the ToleranceZone architecture. Our middleware is located between the sensor network application and the operating system. Each middleware idiom is reflected by a component in the architecture.

Shared Variables: In most applications for WSNs it is necessary to share sensed data with other nodes of the network. For example, in a house control application, were temperature readings of a room are shared with actuators like the heater.

To provide the exchange of data, the ToleranceZone middleware provides an interface to define such shared variables. The middleware takes over the management and updates the variables with a single writer and multiple reader semantic. At first we want to investigate concepts to exchange data within a 1-hop neighborhood of the nodes. In the second step the considered concepts will be extended to a k-hop neighborhood.

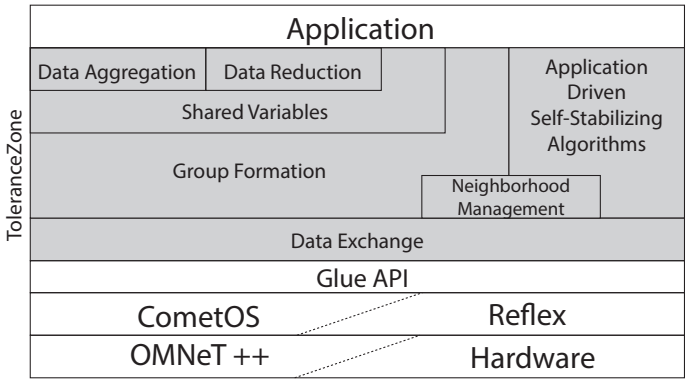

Fig. 1. Architecture of the ToleranceZone Middleware

Because of the behavior of self-stabilization, a first lower bound achieveable is eventual consistency, i.e., we can not give guarantees about individual intermediate states of the shared variables. In most WSN applications, e.g., the heating control, this assumption is sufficient. Still it is possible that applications requires harder restrictions on the deviation they allow, e.g., an emergency stop of a machine caused by overpressure. We will investigate which stronger consistency models can be realized with self-stabilization. Furthermore, we will substantiate if fault detectors can be used to achieve superstabilization as introduced by Dolev [4].

Data Aggregation and Reduction: Based on the shared variables, we are going to investigate self-stabilizing mechanisms for data aggregation and reduction.

In monitoring applications it is often unnecessary to consider all sensed data of the network, in most cases it is sufficient to receive an aggregated value of a time period, for instance, the average temperature of a room. A reduction function can be used to collect the temperature of rooms which have an inadequate heating.

Both functionalities are necessary and common in WSN applications. In the case of self-stabilization, reduction and aggregation is not considered as a one-shot operation. Self-stabilizing reduction as well as aggregation are defined informally: as a continuously updated result, reflecting the latest values of participating nodes.

Group formation: Shared Variables are not necessarily shared within the whole network, neither only with 1-hop neighbors. In sensor network applications, data is often exchanged between nodes fulfilling conjoint tasks. These nodes can be located in different physical locations. Group forming might be based on common properties, e.g., having the same sensors, or manufacturer. Continuing the house control example: The heater does not need the data of the outdoor sprinkling system, but, the intrusion detection needs the data of all motion detectors around the building.

To compute the set of nodes, participating in data exchange, is the task of the group formation component. The

TOLERANCEZONE provides the application developer with an interface to define constraints, that determine, which nodes are chosen for a specific group.

Neighborhood Management: A special group is the neighborhood provided by the neighborhood management component. In case of WSNs, the one-hop communication relation is not known a priori and changes over time. Self-stabilizing algorithms depend on state exchange between nodes within a bidirectional neighborhood, to determine stable system states. Therefore, the neighborhood management component has to discover and to cohere a long-lasting neighborhood. To achieve this, we are investigating additional filters, e.g., link estimators.

At first, the 1-hop neighborhoods is considered. The derived techniques will then be augmented to support bigger, k-hop neighborhoods. This will make more complex self-stabilising algorithems possible.

Additional Elements: The presented components for each idiom fulfill their tasks in a self-stabilizing manner. This makes state exchange of each module necessary. To collect and distribute the state is the main purpose of the *Data Exchange* component. We will investigate mechanisms to avoid unnecessary communication. At a bare minimum, all states will be aggregated to decrease the amount of communication steps. Piggybacking, of certain control data on top of application data, and compression algorithms are promising ideas, which will be evaluated.

The task of the *Application Driven Self-Stabilizing Algorithms* component is to enable developers to run their own self-stabilizing algorithms. To allow this, an interface, to register external rule based algorithms and states, will be provided.

Underlying Systems

Both groups at TUHH and BTU have done previous work in the field of WSNs. To evaluate their work, in simulations and real world deployments, operating systems for embedded systems were developed, these are, COMETOS (TUHH) [10] and REFLEX (BTU) [11] .

Either system is event-driven, component-based, and uses the programming language *C++*. Furthermore, each system supports the *OMNeT++* simulation environment. Making implementation tests, with logging and evaluation, very convenient. Moreover, the direkt portation of simulation code to hardware code is a main feature of both programming abstractions.

GlueAPI: The TOLERANCEZONE middleware is being developed simultaneously for either environment, i.e., COMETOS and REFLEX. Hence, an abstraction layer between the middleware and the operating system is necessary. To achieve this, a set of interfaces was defined called *GlueAPI*. This API provides mechanisms for communication and wraps the event-driven scheduling procedures supplied by the underlying system. The GlueAPI enables the TOLERANCEZONE implementation, thereby the WSN application, to be executed on one of both systems at a time.

Evaluation

The goal of TOLERANCEZONE is to reach a higher fault tolerance compared to traditional approaches. Therefore, several realistic fault scenarios have to be developed to evaluate the difference between the self-stabilizing and the common approach.

For comparison, the *TeenyLime* [12] middleware was chosen. TeenyLime is a suitable, state-of-the-art example of another data centric middleware for WSN. Its tuple-space concept has many similarities to the data exchange of self-stabilizing algorithms.

IV. CONCLUSION

The goal of the TOLERANCEZONE project is to corroborate the following two thesis: First, the common middleware idioms shared variables, data aggregation and reduction, group formation, and neighborhood management, can be realized in an inherent fault-tolerant manner by utilizing self-stabilization. Second, the implementation, based on these self-stabilizing mechanisms, is much more resource-sparing, while reaching at least the same performance as the TeenyLime middleware.

V. ACKNOWLEDGMENTS

ToleranceZone is funded by the Deutsche Forschungsgemeinschaft (DFG NO 625/6-1).

REFERENCES

[1] K. Römer, "Programming paradigms and middleware for sensor networks," Institute for Pervasive Computing, ETH Zurich, Tech. Rep., 2004.

[2] Gummadi, Ramakrishna, Gnawali, Omprakash, Govindan, and Ramesh, "Macro-programming wireless sensor networks using kairos," in *Distributed Computing in Sensor Systems*, 2005.

[3] E. W. Dijkstra., "Self-stabilizing systems in spite of distributed control," in *Communications of the ACM*, 1974.

[4] S. Dolev, *Self-stabilization*. MIT Press, 2000.

[5] T. Herman, "Models of self-stabilization and sensor networks," in *Distributed Computing - IWDC 2003, volume 2918 of Lecture Notes in Computer Science*, 2003.

[6] V. Turau and C. Weyer, "Fault tolerance in wireless sensor networks through," in *Int. J. Commun. Netw. Distrib. Syst.*, 2009.

[7] C. Weyer and V. Turau, "Selfwise: A framework for developing self-stabilizing algorithms," in *Kommunikation in Verteilten Systemen (KiVS), Informatik aktuell*, 2009.

[8] C. Weyer, V. Turau, A. Lagemann, and J. Nolte, "Programming wireless sensor networks in a self-stabilizing style," in *In Third International Conference on Sensor Technologies and Applications*, 2009.

[9] S. Lohs, R. Karnapke, J. Nolte, and A. Lagemann, "Self-stabilizing sensor networks for emergency management," in *Second Workshop on Pervasive Networks for Emergency Management*, 2012.

[10] S. Unterschütz, A. Weigel, and V. Turau, "Cross-platform protocol development based on omnet++," in *In Proceedings of the 5th International Workshop on OMNeT++ (OMNeT++'12)*, 2012.

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

[12] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "Teenylime: Transiently shared tuple space middleware for wireless sensor networks," in *In Proceedings of the 1st ACM International Workshop on Middleware for Sensor Network (MIDSENS - colocated with ACM/IFIP/USENIX Middleware)*, 2006.