Simplified Programming of Faulty Sensor Networks via Code Transformation and Run-Time Interval Computation

Lan S. Bai† Robert P. Dick† Peter A. Dinda‡ Pai H. Chou*
{lanbai, dickrp}@umich.edu pdinda@northwestern.edu phchou@uci.edu
†University of Michigan ‡Northwestern University *University of California, Irvine

***Abstract*—Detecting and reacting to faults is an indispensable capability for many wireless sensor network applications. Unfortunately, implementing fault detection and error correction algorithms is challenging. Programming languages and fault tolerance mechanisms for sensor networks have historically been designed in isolation. This is the first work to combine them. Our goal is to simplify the design of fault-tolerant sensor networks. We describe a system that makes it unnecessary for sensor network application developers and users to understand the intricate implementation details of fault detection and tolerance techniques, while still using their domain knowledge to support fault detection, error correction, and error estimation mechanisms. Our FACTS system translates low-level faults into their consequences for application-level data quality, i.e., consequences domain experts can appreciate and understand. FACTS is an extension of an existing sensor network programming language; its compiler and runtime libraries have been modified to support automatic generation of code for on-line fault detection and tolerance. This code determines the impacts of faults on the accuracies of the results of potentially complex data aggregation and analysis expressions. We evaluate the overhead of the proposed system on code size, memory use, and the accuracy improvements for data analysis expressions using a small experimental testbed and simulations of large-scale networks.**

I. Introduction

This work is part of a project that aims to develop easy-to-use programming languages for novice programmers in order to open sensor network design to application experts, such as geologists, biologists, and farmers, who generally have little programming experience.

Programming a wireless sensor network is essentially the design of a resource-constrained distributed software system operating on fault-prone sensor nodes and wireless links. It can therefore require expertise in distributed embedded system design. High-level sensor network programming languages [1]–[6] can simplify the design of sensor networks; programmers can ignore low-level implementation details and focus on application-level logic. Our previous study has demonstrated that with a compact and high-level language, even novice programmers can specify real sensor network applications correctly and efficiently [7], [8]. Although many sensor network programming languages spare designers from explicitly describing low-level details such as data transmission and network topology management, support for fault detection and error correction is seldom considered.

It is important for a sensor network program to be capable of detecting and reacting to faults. Sensor nodes are composed of fault-prone components and they often operate in harsh environments. Experience from prior deployments [9]–[11] has demonstrated that deployed nodes can fail or produce erroneous results. A fault is an incorrect state of hardware or software resulting from failures of components, physical interference from the environment, or incorrect design. For example, a sensor node may experience a fault when water leaks through its package and damages sensors. A system failure occurs when faults prevent the system from providing a required service. Embedding fault detection, fault recovery, and error estimation functionalities in a program makes it more robust and allows more accurate interpretation of data gathered by the application.

Many researchers have proposed methods for fault detection and fault correction for sensor networks [12]–[17]. These technologies may be readily used by experienced sensor network developers. However, it requires tremendous efforts for novice programmers to learn and use these techniques, especially in the context of wireless sensor network design.

I.A. Goals and Contributions

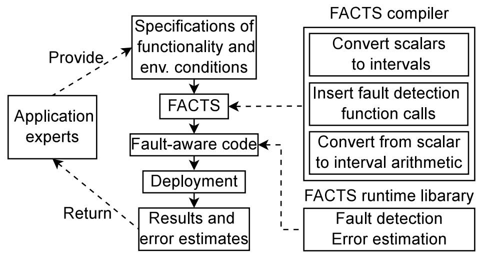
Reliability is a central concern for wireless sensor networks, and developing fault-tolerant distributed applications is challenging, especially for those with professions other than software engineering, i.e., most people with a need for distributed sensing. Our goal is to combine high-level programming languages and automatic fault-aware code transformation techniques to empower novice programmers to develop sensor networks that can operate reliably, potentially in harsh environments. Our work is based on three insights. (1) Novice programmers tend to assume a fault-free system during programming. (2) Application experts care primarily about application-level performance; they should be informed about the impact of errors on the end products of an application, but reporting every detail about low-level sensor network faults imposes great burden with little value. (3) The knowledge of application experts about expected behaviors and environmental conditions can be used to allow more effective fault detection and correction.

We have designed, implemented, and evaluated a system, called FACTS (Fault-Aware Code Transformation for Sensor networks), to simplify programming faulty sensor networks. FACTS hides faults from programmers but indicates the impact of low-level faults on application outputs, i.e., the end results of data processing expressions in the application specification. We implemented FACTS by extending an existing high-level programming language for sensor networks. The current design of FACTS focuses on data-acquisition applications and sensor data faults. Programmers provide specifications of application logic as well as of expected environmental conditions. The compiler automatically generates fault-aware code to which fault detection, error correction, and error estimation functionalities have been added. During network operation, sensor faults are detected by identifying sensor readings that fall outside of application-specific ranges. In case of sensor faults, the ranges of actual data values are estimated using temporal and spatial correlation. The ranges of end results produced via potentially complex expressions are then computed.

Our work makes three main contributions.

1) We describe an approach to simplify programming of potentially faulty sensor networks by automatically generating code for fault detection, error correction, and error estimation.
2) We develop an error estimation technique to calculate the error bounds for application data as a result of faults.

This work was supported by the National Science Foundation under awards CNS-0721978, CNS-0910816, and CNS-0347941.



FACTS compiler
Specifications of functionality and env. conditions
Provide
Application experts
FACTS
Fault-aware code
Deployment
Return
Results and error estimates
Convert scalars to intervals
Insert fault detection function calls
Convert from scalar to interval arithmetic
FACTS runtime libarary
Fault detection
Error estimation

Fig. 1. Overview of FACTS system.

3) We implement this approach in a real system by modifying the compiler and runtime library for a high-level sensor network programming language.

We evaluate the overhead of our system on code size, memory use, and improvement in end result accuracy using a small-scale testbed and simulation of larger-scale networks. The average code size overhead is 15% and the average memory overhead is 3.6%. The resulting intervals produced by FACTS always contain the actual value, while the fault-unaware program can produce substantial errors. Interested readers can learn more about FACTS at our project website [18].

I.B. Related Work

Numerous high-level programming languages have been proposed for wireless sensor networks to ease their development process. These languages share a common strategy: they provide appropriate abstractions to hide low-level implementation details from programmers. For example, macro-programming languages let programmers treat the whole network as a single machine, thus hiding node-level communication details from programmers [1]–[4]. Domain-specific languages target a certain type of application and provide commonly used algorithms in their libraries [19]. Other languages [5], [6] based on commercial data processing tools let programmers describe how data are processed without concerning themselves with the details of data gathering. Though these languages have reduced programming complexity compared to node-level programming languages, none provide support for fault detection and error correction. Even if programmers are willing to deal with greatly increased implementation complexity, some macroprogramming languages do not provide node-level communication primitives, making it intractable for the programmer to implement fault detection and correction techniques. To the best of our knowledge, only one sensor network language explicitly supports fault tolerance [20]. It provides declarative annotations to specify checkpointing recovery strategies. In contrast, our system (FACTS) does not require programmers to explicitly deal with faults, making it accessible even to sensing application experts with limited programming experience. The approach used in FACTS can also be applied to other languages.

Bai et al. developed a language called WASP that allows novice programmers to specify periodic data collection applications [7]. The application implementation success rates and development times of novice programmers using this, and alternative sensor network programming languages have been experimentally evaluated. Unfortunately, WASP did not support fault modeling or management. Faulty sensor readings have the potential to distort aggregated results. Worse yet, allowing users to concern themselves with only the end results of data processing makes it less likely that sensor-level faults will be noticed. We implemented FACTS by extending WASP, as well as its compilation and runtime system, to support code transformations for fault tolerance.

Researchers have identified and classified various types of faults in sensor networks and proposed numerous approaches for fault detection [12], tolerance [13], [14], diagnosis [15], [16], and recovery [17], [20]. These papers concentrate on minimizing the impact of faults on system performance and availability. Our goal is to support reliability management techniques without requiring programmers to understand the their implementation details. We also propose an error estimation technique to provide application experts with a more accurate and informative view of data gathered from a network.

TABLE I
EXAMPLE OF FAULT CORRECTION

	Node 1	Node 2	Node 3	Average
True value (°C)	10	12	14	12
Sensor reading (°C)	10	12	0	7.3
Corrected reading (°C)	10	12	[10, 16]	[10.7, 12.7]

II. FAULT-AWARE CODE TRANSFORMATION FOR SENSOR NETWORKS (FACTS)

We now describe the FACTS architecture.

II.A. FACTS System Architecture

The purpose of FACTS is to shift responsibility for the mechanical aspects of fault management from programmers to the programming language, compiler, and run-time libraries. In this paper, we focus on data acquisition applications. The left hand side of Figure 1 illustrates how application experts use our system. An application expert specifies application functionality and expected environmental conditions. FACTS uses this information to generate an implementation that is capable of fault detection, fault recovery, and error estimation. The application expert then deploys a network running the generated code. FACTS indicates the application-level impact of faults, i.e., the error range for the end results of potentially complex data processing expressions, while hiding component-level implementation and fault details from the application expert.

The right hand side of Figure 1 shows the system components and their purposes. The original compiler generates node-level code that implements sensing, data transmission, and data aggregation algorithms. FACTS provides a runtime library to detect faults and estimate errors. The FACTS compiler modifies and augments the original compiler in the following ways to generate fault-aware code. (1) It changes the types of some variables in the program to include extra information about error estimates. (2) It transforms arithmetic expressions to interval arithmetic expressions so the implications of faults can be propagated to the end results. (3) It inserts calls to fault detection and error estimation functions.

We now use an example to demonstrate the key ideas of our approach. Consider the application that monitors redwood tree microclimates [21]. Biologists deploy sensor nodes on a redwood tree to gather temperature and humidity data. The application periodically samples temperature and humidity, averages readings from nodes at similar heights, and sends the results to a base station. Assume at one height, there are three nodes with identifiers 1, 2, and 3. Table I shows an example of data gathered during one sampling cycle. The second row shows the ground truth values for each node. The third row shows the sensor readings. Node 3 is faulty: the fault results in an erroneous sensor reading of 0 °C. Without any fault tolerance mechanisms, the average of the three values in the second row, 7.3 °C, is returned to the user. Unfortunately, the user is unaware that 7.3 °C is an erroneous result that underestimates the average temperature by 4.7 °C.

With FACTS, the expert designing the application provides some information about the environment in a simple format. For example, the expected temperature range is 10–30 °C. The code generated by FACTS uses this information to detect the fault at node 3. Instead of using the incorrect value of 0 °C, it indicates that the value is in the range 10–16 °C based on historical readings and readings from other nearby nodes. FACTS then propagates this interval through the expressions to produce the value of interest for the network (i.e., the average), indicating that it is in the range 10.7–12.7 °C. The user is made aware of the system-level implications of the low-level fault. This error information can be further used by application experts during their data analysis and help them draw more accurate scientific conclusions. The actual techniques used in FACTS are more sophisticated than those considered in this explanatory example. For example, FACTS considers the influence of spatial and temporal correlation as well as the impact of expressions predicated on faulty variables.

Our approach has the following features.

1) Application experts do not need to understand the intricacies of sensor network faults or explicitly manage them.
2) The domain knowledge of application experts is used to allow fault detection and error estimation, without imposing much additional specification burden.
3) System-level error bounds are provided to application experts to allow more thorough understanding of data.

II.B. Specification of Environmental Conditions

Application experts' knowledge of environmental conditions can be used for two purposes: detecting sensor faults and correcting for faulty sensor readings. Sensed data characteristics can be determined based on sensor specifications and environmental conditions such as data value range, temporal gradient (change in value per time unit), temporal correlation, spatial variance (change in value per distance unit), and spatial correlation. In this work we use range, maximum temporal gradient, and maximum spatial gradient to describe the environmental conditions; however, these concepts can be extended to use other parameters. We extend WASP programming language to let programmers specify an expected range and maximum temporal/spatial gradient for each environmental parameter. After the programmer provides the application specification, a list of relevant physical parameters is extracted to produce a template for the programmer to input information about their expected behaviors. The programmer need only read the template and enter a few numbers.

II.C. Fault Detection and Sensor Error Estimation

In this work, we focus on methods that can be implemented efficiently in software and detect a commonly occurring class of faults. Although the proposed error estimation technique will work with any hardware or software fault detection mechanism, we use the following detection criteria in our FACTS system prototype: (1) are the sensor data within the expected range? and (2) are the environmental conditions within the operating range of the sensors?

It is common for faulty sensor nodes to produce abnormal readings. For example, developers of a habitat monitoring network observed abnormally large or small sensor readings (light, temperature, and humidity) when water penetrated the enclosure of the sensor node and affected the power supply [10]. Developers of a redwood tree macroclimate monitoring network associated out-of-range sensor readings with node faults caused by a drop in battery voltage [21]. Such faults can be detected via range checking.

As sensors cannot work properly in certain environmental conditions, sensor faults can also be detected by checking whether the current environmental conditions are within the sensor's operating range. If either requirement is violated, the sensor reading is deemed incorrect. Consider an application that gathers light level readings using TelosB sensor nodes. The S1087 light sensor on TelosB nodes has an operating temperature range of -10–60 °C. Both the light and temperature sensor readings are checked to detect light sensor data faults. A fault is likely to occur if either the light sensor reading is out of the expected range or the temperature sensor reading is out of the -10–60 °C range. Note that when an undetected fault occurs in the temperature sensor, we may (conservatively but sometimes mistakenly) deem the light sensor to be faulty. Faults in sensors on the same node may be correlated because they share many hardware and software components; the developers of the habitat monitoring sensor network observed this correlation [10]. Therefore, the false positive rate due to faults in the sensor monitoring the operating environment is likely to be lower than would be the case in the absence of sensor fault correlation.

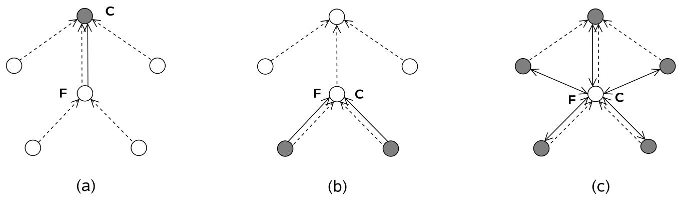


Fig. 2. Design options for error estimation based on spatial data.

Local error estimation is used to indicate the intervals of actual data elements and expressions when faults are detected. Faulty sensor readings are estimated based on bounds on environmental parameters and their spatial and temporal gradients. Data gathered from a sensor network often change gradually with time and location. Temporal and spatial variations can be bounded for many applications. We use such bounds to replace erroneous values with ranges. For example, given a maximum temporal gradient for temperature of 1 °C per minute, the range of a faulty temperature reading can be estimated as 19–21 °C if the most recent correct reading of 20 °C was taken one minute ago. In other words, in case of an erroneous reading, the possible temperature range is estimated based on other data. The FACTS compiler creates data buffers to store historical data. The buffer size is determined based on the user-specified bounds and sampling periods. For example, if the temporal variation of temperature is at most 5 °C over one hour and the temperature sampling period is 10 minutes, then a buffer of size 6 is used.

A bound on spatial gradient indicates the maximum change per meter. Error estimation using spatial gradients requires knowledge of distances between sensor nodes. If node locations are known at design time, the locations of nearby nodes can be stored in a table and used for error estimation at runtime. If node locations are unknown until deployment, distances between nodes can be estimated using node localization algorithms [22].

Error estimation using spatial gradients requires data from other nodes and may therefore introduce communication overhead. The locations where the implications of faults are estimated and the amount of spatial data used impact energy overhead and the tightness of the resulting error bounds. Figure 2 shows examples of three design options for a network composed of six nodes. A dotted arrow represents a communication link in the routing tree, originating from a child node and ending at a parent node. **F** indicates where a fault occurs and **C** indicates where the error resulting from this faulty reading is estimated. The center node has a faulty sensor reading. To simplify explanation, we ignore error estimation based on temporal changes in this example. Sensor data from the shaded nodes are used to estimate the interval for incorrect readings gathered by the faulty sensor. A solid arrow indicates the links on which the corresponding design option produces communication overhead.

In design (a) (in Figure 2), the parent node estimates the value interval for the faulty node by based on the parent node's sensor value. In design (b), the value interval at the faulty node is estimated using its children's readings. In design (c), the value interval at the faulty node is estimated using all its neighbors' readings. Design (a) uses only one neighboring node (parent node) for estimation, design (b) uses all children nodes, and design (c) uses all neighboring nodes. Design (a) imposes communication overhead on the link from a faulty node to its parent, design (b) requires every node to always send its own raw sensor readings to its parent, design (c) requires the faulty node to broadcast requests to which its neighbor nodes reply with their sensor readings. The more information used in estimating an interval, the tighter the bound is; design (a) provides the loosest bound with the lowest communication overhead and design (c) provides the tightest bound with the highest communication overhead. We choose design (b) in the FACTS system implementation. This option requires the least modification to the network protocol, and supports the use of multiple spatial readings for error estimation. Specifically, each node sends the aggregated results of the subtree it is the root for and its own raw sensor reading.

II.D. Error Propagation

The WASP programming language supports node-level data processing functions and network-level aggregation functions. FACTS computes the errors in expression results based on the sensor readings they depend on. Specifically, FACTS returns estimated ranges associated with each requested datum, i.e., every data element in the `COLLECT` statement for network-level data gathering and aggregation in a WASP program. As described in Section II-C, faulty sensor readings are replaced with estimated ranges. The errors are then propagated to final results using interval arithmetic. Error estimation for network-level aggregation results can occur either in the network or at the base station. The former approach aggregates correct and faulty variables in the network and estimates the associated error. The latter approach aggregates only correct variables in the network and forwards faulty variables to the base station, where the error of the final results are computed. We adopt the former approach in FACTS because it implies smaller data transmission overhead and scales with network size and fault rate.

Errors caused by faulty sensor readings can propagate to end results via mathematical operations such as addition. The error estimation problem can be defined as follows. Given $y = f(x_1, x_2, \cdots, x_n)$ and the range of each x_i, estimate the range of y. Each x_i represents a potentially erroneous variable. y represents the returned result. This can be solved with interval arithmetic [23], in which arithmetic operations are applied to operand intervals to calculate result intervals. Interval arithmetic has been applied to rounding error estimation and circuit timing analysis. In contrast to these uses, maintaining low overhead is more important for our (on-line) application because error estimation may execute on energy- and time-constrained sensor nodes. Fortunately, for the built-in functions supported by the WASP programming language, it is easy to find the range of an output given ranges of inputs. For example, the frequently used aggregation functions such as `MAX`, `MIN`, and `AVG` are all monotonic, allowing the extremes of an output to be computed directly by applying the operation on extremes of inputs. The mathematical expressions and functions used in the majority of published wireless sensor network deployments can also be efficiently computed following interval arithmetic rules.

Errors can also propagate to end results via their influence on control flow. When a faulty variable is used in a predicate expression and its estimated range spans the predicate threshold, the range of the result is computed by combining the ranges that would result from either branch. For non-aggregating data collection applications, the predicates determine whether data should be sent to the base station. For applications with in-network spatial data aggregation, the predicates determine whether data should be included in the network-level aggregation operations. For example, `COLLECT AVG(y) WHERE x > 100` requires the y variable of a particular node to be included in the averaging operation if that node's x value is above 100. When a fault results in the interval for x spanning 100, the range of interval `AVG(y)` (the average of all node y values) should span the results calculated with and without including y from the faulty node. The error estimation problem can more formally be defined as follows.

Given that $y = f(x_1, x_2, \cdots, x_n)$ where f is an aggregation function for which x_i may or may not be included in the argument list and n is number of variables, compute the range of y. The range of y can be naïvely obtained by computing the ranges for the 2^n cases separately and calculating their union. The computational complexity may be acceptable for a sparse network since n is bounded by the maximum number of immediate children nodes. Fortunately, for the aggregation operations commonly used in wireless sensor network deployments, the computational complexity is less than $\mathcal{O}(n \log n)$. For `MAX` and `MIN` operations, including one more argument only monotonically affects the upper or lower bound of the result. Therefore, the range of the result can be easily calculated by iteratively considering each of the n variables. For `AVG`, adding one more variable may increase or decrease both lower and upper bounds. However, the range of the result can still be computed in $\mathcal{O}(n \log n)$ time. For example, to get the lower bound of the `AVG` result, first order the lower bounds of the intervals associated with the nodes that may meet the selection requirements (but are not certain to meet them) in increasing order. Incrementally scan the ordered list, add each value to the set of values to average, and recompute the result until the local minimum for the result is reached. To get the upper bound of the `AVG` result, use a similar technique, but instead scan the upper bounds of the intervals in decreasing order. Consider the expression `COLLECT AVG(y) WHERE x > 100` as an example. Assume a network of five nodes. Their x ranges are [120], [90, 110], [80, 120], [83, 102], and [130]. Their y values are 2, 4, 6, 3, and 8. To compute the upper bound of `AVG(y)`, we order the y values except 2 and 8 (they must be involved in computing the average) in descending order. The average of 2 and 8 is 5. After including 6, the average becomes 5.3. After including 4, the average becomes 5. Therefore, the upper bound for `AVG(y)` is 5.3.

II.E. Automated Code Transformation

We now describe a software implementation to support automatic online fault detection and error estimation. The following steps will be used to generate fault-aware code. (1) Replace sensor readings and the variables that depend upon them with tuples containing two variables of the same type. (2) Insert calls to fault detection functions after sensor readings are obtained. The fault detection methods have been described in Section II-C. (3) Insert calls to temporal gradient-based error estimation functions after error detection function calls to calculate ranges for faulty variables. (4) Insert calls to spatial gradient based error estimation functions before a node aggregates its received data. (5) Convert mathematical expressions involving possibly faulty variables to interval arithmetic operations. For example, $z = x + y$ is converted to $z.low = x.low + y.low; z.high = x.high + y.high$.

III. Experimental Evaluation

We evaluated the accuracy of the value estimates provided by FACTS, as well as its impact on code size and memory use. Our evaluation uses a small-scale experimental hardware testbed and simulations of a larger-scale network composed of 74 sensor nodes with real-world data traces. This section describes the experimental setup and the results.

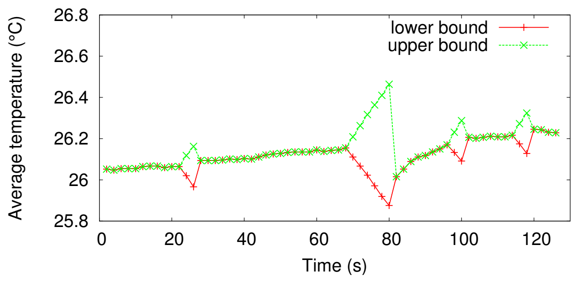


Fig. 3. Temperature interval as a function of time.

TABLE II
FAULT-AWARE AND UNAWARE IMPLEMENTATIONS

	Code size (B)			Memory usage (B)		
	App.1	App.2	App.3	App.1	App.2	App.3
Fault-unaware	32,556	33,060	27,722	2,130	2,134	2,038
Fault-aware	37,358	37,740	32,088	2,212	2,224	2,096
Overhead (%)	14.7	14.2	15.7	3.8	4.2	2.7

III.A. *Prototype Evaluation*

We implemented a prototype of the proposed system and tested it in a small-scale sensor network consisting of four TelosB nodes. Each node samples temperature every 2 s. The average across all nodes is returned to the base station. The results contain a tuple for each sampling cycle indicating the upper and lower bounds on the average temperature. Figure 3 displays the temperature upper and lower bounds as functions of time. We injected intermittent sensor faults by shorting the terminals of the thermal sensor to produce readings of -39.6 °C (from a 0 V analog-to-digital converter input), e.g., at 22 s. In the absence of faults, the upper and lower bounds in Figure 3 are identical. The estimated bounds become looser over time when the intermittent fault persists, due to the use of temporal correlation to calculate the temperature interval. This section serves primarily to demonstrate that a functioning prototype of the FACTS system has been implemented, and explain its operation.

III.B. *Evaluation of Code Size and Memory Use Overhead*

To evaluate the impact of using FACTS on code size and memory requirements, we compared the code generated with the original WASP compiler and the extended FACTS compiler. Table II shows the code size and memory use for the three representative examples based on deployed sensor network applications [7]. Application 1 periodically gathers temperature and light data. Application 2 periodically samples light and averages data among nodes at similar heights. Application 3 periodically samples temperature and sends data only when the increase in temperature exceeds a threshold. The average code size overhead across the three applications is 15% and the average memory overhead is 3.6%.

We compare the lines of code for the high-level specification input to FACTS as well as the generated node-level code to give some evidence of its impact on programming complexity. The results are shown in Table III. The applications are the same as those used for code size and memory use estimation. FACTS only requires three to six additional lines of code in the high-level specification, depending on how many physical parameters are sensed. Note that the syntax of

TABLE III
LINES OF CODE FOR FAULT-AWARE AND UNAWARE IMPLEMENTATIONS

	High-level specification			Emitted NesC code		
	App.1	App.2	App.3	App.1	App.2	App.3
Fault-unaware	6	7	7	489	495	484
Fault-aware	12	10	10	621	585	545

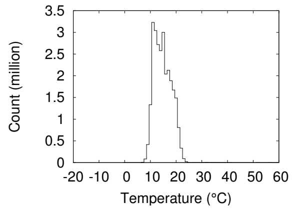


Fig. 4. Temperature histogram.

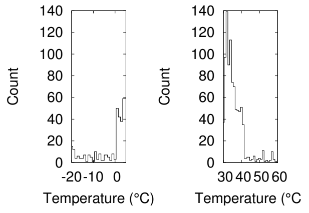


Fig. 5. Outlier histograms.

the FACTS specifications is at least as simple as that for functionality. Therefore, we argue that the programming complexity only increases slightly. Given that researchers have previously demonstrated via user studies that novice programmers can use WASP correctly and efficiently [7], and the additional specifications required by FACTS have low complexity and length, we believe that the FACTS system will remain accessible to novice programmers. In contrast, the low-level NesC code (excluding library code) increases in length by 61, 90, or 132 lines of code depending on application. This implies that the extra programming efforts required to manually and explicitly handle sensor faults is potentially high. With FACTS, the increased implementation complexity is not exposed to programmers.

III.C. *Simulation of Large-Scale Network to Evaluate Impact of Varying Fault Rates*

Simulation environment: We use the SIDnet-SWANS simulator [24] and temperature measurement time series from a real network deployment [9] to model a network of 74 nodes that sample temperature every 29.3 seconds and aggregate data in the network. We assume two aggregation expressions: average and minimum.

Environmental data generation: We use the data from the LUCE deployment at the EPFL campus [9] to provide environmental data for our simulation. The LUCE deployment contains 97 weather stations that span a 500 m×300 m area and ran for 6 months. We take the following steps to generate fault-free data traces from the original data set. (1) It is important that faults be rare in our input data so that we can determine the actual ground truth data values with which our fault correction system ranges and estimates will be compared. We identified a time interval in which the data drop rate from most of the nodes is small and eliminated from consideration 23 nodes that have high drop rates in that time interval. We used a one-hour trace containing 9,028 data samples from 74 nodes. (2) The original data set has a period of 29.3 s with small jitter. We parse the data to produce synchronized periodic time series. Multiple samples associated with the same period are averaged, while periods without data are recovered by selecting from the valid data value distribution for the application. Combined, these faults only affected 3.7% of the time series data. (3) We determine the lower and upper bounds based on the histogram of temperature data from the 74 nodes. The histogram is shown in Figure 4, where 99.4% of the data are in the 5–30 °C range. Inspection indicates that data outside this range are associated with spikes in the time series. We treat data outside this range as outliers. (4) We analyze the temporal and spatial correlation ignoring outliers and compute the bounds on temporal and spatial gradients. The results are 3 °C per 29.3 s and 5 °C per 50 m. (5) We replace faulty and missing data (only 3.7% of the original data traces) with values that are generated based on spatial and temporal correlation. The resulting data set complies with the bounds on data range, temporal gradient, and spatial gradient.

Fault injection during evaluation: We model sensor transient faults using Poisson processes, primarily because many transient fault processes are memory-less. This influences only our simulation results, not the design of the FACTS system, which can handle fault processes with arbitrary temporal density functions. We use independent but

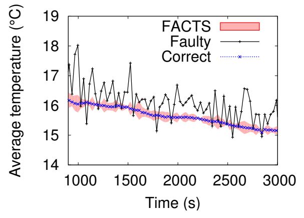


Fig. 6. Results with and without FACTS.

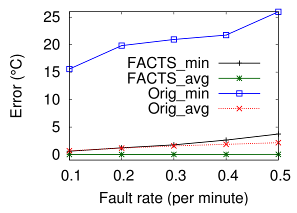


Fig. 7. Dependence of error (in °C) on fault rate.

equal-rate fault processes for different sensor nodes. Faulty sensor readings are generated by sampling from the set of outliers extracted from the original data set; this was done so that the simulated and actual faulty data would have the same distribution, which is shown in Figure 5. We run simulations with multiple fault rates, ranging from 0.1 to 0.5 per minute, to study the impact of fault rate on accuracy. The tested fault rates are selected based on a survey on sensor network data faults by Ramanathan et al. [16]. The sensor fault durations in the original data are generally less than 29.3 s (one sampling cycle), supporting the injection of transient faults. For each fault rate, we run 5 simulations with different random seeds and average the results.

Results: Figure 6 shows an example simulated time series for a fault rate of 0.1 per minute. The shaded area shows the value intervals produced by FACTS. The curve inside it shows ground truth results from the original time series. The figure shows that the original fault-unaware program can produce substantial errors (0.7 °C on average and 2.7 °C maximum) and that the intervals produced by FACTS always contain the actual value.

If the midpoints of the intervals produced by FACTS are used as value estimates, the error relative to ground truth values can be computed. Figure 7 shows aggregate error for simulation runs with different fault rates. The root mean square errors relative to the ground truth data are computed for the fault-aware and fault-unaware programs. FACTS_min and FACTS_avg represent the results for the minimum expression application and the average expression application. Orig_min and Orig_avg represent the results for the fault-unaware versions of these applications. FACTS results have an average error of 0.02 °C and fault-unaware results have an average error of 3.86 °C. The worst-case errors are 5.95 °C and 36.40 °C for FACTS and the fault-unaware systems.

IV. Conclusions

Detecting and reacting to faults are important capabilities for many sensor network applications. Requiring application experts to explicitly program fault detection and error estimation algorithms imposes implementation burden and increases the probability of introducing software errors. We have described an approach to simplify fault detection and management in wireless sensor networks. This approach, called FACTS, is designed to be accessible to application experts who may not be expert programmers. Users of FACTS only need to specify high-level application functionalities and expected environmental conditions. FACTS is implemented by extending an existing high-level sensor network language, compiler, and run-time system. It uses easily specified domain-specific expert knowledge to support on-line detection of some classes of sensor faults. When faults are detected, FACTS adjusts the accuracy intervals of data analysis expressions to make the system-level impact of faults clear to sensor network users. A small-scale hardware testbed and simulations of a 74-node network using real-world sensor data show that FACTS substantially increases estimation accuracy and imposes little overhead compared to fault-unaware programs.

References

[1] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Trans. Database Systems*, vol. 30, no. 1, pp. 122–173, Mar. 2005.
[2] R. Müller, G. Alonso, and D. Kossmann, "SwissQM: Next generation data processing in sensor networks," in *Proc. Conf. on Innovative Data Systems Research*, Jan. 2007, pp. 1–9.
[3] U. Bischoff and G. Kortuem, "A state-based programming model and system for wireless sensor networks," in *Proc. Int. Conf. on Pervasive Computing and Communications Wkshp.*, Mar. 2007, pp. 261–266.
[4] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proc. Int. Conf. Information Processing in Sensor Networks*, Apr. 2007, pp. 489–498.
[5] M. Ghercioiu, "A graphical programming approach to wireless sensor network nodes," in *Proc. Sensors for Industry Conf.*, Feb. 2005, pp. 118–121.
[6] J. Horey, E. Nelson, and A. B. Maccabe, "Tables: A table-based language environment for sensor networks," The University of New Mexico, Tech. Rep., 2007.
[7] L. S. Bai, R. P. Dick, and P. A. Dinda, "Archetype-based design: sensor network programming for application experts, not just programming experts," in *Proc. Int. Conf. Information Processing in Sensor Networks*, Apr. 2009, pp. 85–96.
[8] J. S. Miller, P. A. Dinda, and R. P. Dick, "Evaluating a BASIC approach to sensor network node programming," in *Proc. Conf. on Embedded Networked Sensor Systems*, Nov. 2009, pp. 155–168.
[9] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *Proc. Int. Conf. Embedded Networked Sensor Systems*, Nov. 2008, pp. 43–56.
[10] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler, "Lessons from a sensor network expedition," in *Proc. European Wkshp. on Sensor Networks*, Jan. 2004.
[11] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture," in *Proc. Int. Wkshp. Parallel and Distributed Real-Time Systems*, Apr. 2006, pp. 1–8.
[12] F. Koushanfar, M. Potkonjak, and A. Sangiovanni-Vincentelli, "On-line fault detection of sensor measurements," in *Proc. Int. Conf. on Sensors*, Oct. 2003, pp. 974–980.
[13] T. Clouqueur, K. K. Saluja, and P. Ramanathan, "Fault tolerance in collaborative sensor networks for target detection," *IEEE Transactions on Computers*, vol. 53, no. 3, pp. 320–333, Mar. 2004.
[14] H. Liu, P.-J. Wan, and X. Jia, *Fault-tolerent relay node placement in wireless sensor networks*. Springer, Aug. 2005, pp. 230–239.
[15] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong, "Passive diagnosis for wireless sensor networks," in *Proc. Int. Conf. Embedded Networked Sensor Systems*, Nov. 2008, pp. 113–126.
[16] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proc. Int. Conf. Embedded Networked Sensor Systems*, 2005, pp. 255–267.
[17] K. Ni, N. Ramanathan, M. N. H. Chehade, L. Balzano, S. Nair, S. Zahedi, E. Kohler, G. Pottie, M. Hansen, and M. Srivastava, "Sensor network data fault types," *ACM Trans. on Sensor Networks*, vol. 5, no. 3, pp. 1–29, May 2009.
[18] 2009, http://absynth-project.org.
[19] K. Chintalapudi, J. Paek, O. Prakash, T. Fu, K. Dantu, J. Caffrey, R. Govindan, and E. Johnson, "Structural damage detection and localization using NETSHM," in *Proc. Int. Conf. Information Processing in Sensor Networks*, Apr. 2006, pp. 475–482.
[20] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan, "Declarative failure recovery for sensor networks," in *Proc. Int. Conf. Aspect-oriented software development*. ACM, Mar. 2007, pp. 173–184.
[21] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, "A macroscope in the redwoods," in *Proc. Int. Conf. Embedded Networked Sensor Systems*, Nov. 2005, pp. 51–63.
[22] A. Savvides, W. Garber, S. Adlakha, R. Moses, and M. B. Srivastava, "On the error characteristics of multihop node localization in ad-hoc sensor networks," in *Proc. Int. Wkshp. Information Processing in Sensor Networks*, Apr. 2003, pp. 317–332.
[23] R. Moore, *Interval Analysis*, G. Forsythe, Ed. Prentice-Hall, 1817.
[24] O. C. Ghica, G. Trajcevski, P. Scheuermann, Z. Bischof, and N. Valtchanov, "SIDnet-SWANS: a simulator and integrated development platform for sensor networks applications," in *Proc. Int. Conf. Embedded Networked Sensor Systems*, Nov. 2008, pp. 385–386.