A Run-timE, Distributed, Flexible, Lightweight, And Generic Fault Detection Service for Data-Driven Wireless Sensor Applications

Iñigo Urteaga†, Kevin Barnhart‡ and Qi Han†

†Dept. of Mathematical and Computer Sciences, ‡Div. of Environmental Science and Engineering
Colorado School of Mines, Golden, CO 80401
iurteaga@mines.edu, kbarnhar@mines.edu, qhan@mines.edu

Abstract—Increased interest in Wireless Sensor Networks (WSNs) by scientists and engineers is forcing WSN research to focus on application requirements. Data is available as never before in many fields of study; practitioners are now burdened with the challenge of doing data-rich research rather than being data-starved. In-situ sensors can be prone to errors, links between nodes are often unreliable, and nodes may become unresponsive in harsh environments, leaving to researchers the onerous task of deciphering often anomalous data. Presented here is the REDFLAG fault detection service for WSN applications, a Run-timE, Distributed, Flexible, detector of faults, that is also Lightweight And Generic. REDFLAG addresses the two most worrisome issues in data-driven wireless sensor applications: abnormal data and missing data. REDFLAG exposes faults as they occur by using distributed algorithms in order to conserve energy. Simulation results show that REDFLAG is lightweight both in terms of footprint and required power resources while ensuring satisfactory detection and diagnosis accuracy. Because REDFLAG is unrestrictive, it is generically available to a myriad of applications and scenarios.

I. INTRODUCTION

WSNs are resource constrained and are often deployed in harsh environments. As a result, nodes may become frequently inaccessible due to poor connectivity, faulty hardware, energy-supply limitations or environmental threats, leading to a worrisome amount of erroneous or missing data [1], [2]. Therefore, although WSN technology promises to make accessible a vast quantity of information, it poses huge data analysis and management challenges for data-driven sensor applications.

Motivation for this fault detection service comes from subsurface contaminant plume monitoring using a WSN (e.g., [3], and [4]). This data-sensitive application does not have the benefit of a dense network topology, data collection may be event-driven, and there may not exist resource-plentiful nodes in the network. These characteristics can also be found in many other WSN applications. Presented here is REDFLAG, a Run-timE, Distributed, Flexible, fault detection service, that is also Lightweight And Generic. REDFLAG aims to reduce data uncertainty in data-driven wireless sensor applications. It is comprised of a Sensor Reading Validity (SRV) subservice, which detects erroneous sensor readings, and a Network Status Report (NSR) subservice, whose task is to abate data loss by identifying unresponsive nodes. Together, they expose faults as they occur (i.e., at run-time) by using distributed algorithms which offer many tunable parameters (hence, flexible) to the application. The service is also lightweight both in terms of footprint and required power resources. Because REDFLAG is unrestrictive, it is generically available to a myriad of applications and scenarios.

REDFLAG is distinctly different from previously proposed WSN fault detection schemes. Often times, methodologies (e.g., [5]) require a dense network to ensure spatio-temporal relationships in the data, while others expect the measured phenomenon to follow relatively simple models (e.g., [6]). Other systems (e.g., Sympathy [7], SNMS [2]) are generally centralized, where diagnosis data is periodically reported to a centralized server, requiring more powerful stations. Still, others (e.g., [8]) address various networking faults in isolation, such as power depletion, hidden terminals, congestion detection and mitigation, and asymmetric communication links. In contrast, REDFLAG reduces data uncertainty by detecting network faults and by identifying the root causes of missing or abnormal sensor readings, the two most common issues in data-intensive wireless sensor applications.
REDFLAG resides in each node and it is designed from a layered point of view (Fig. 1). REDFLAG runs in a collaborative manner to detect and diagnose faults in a WSN before reporting alarms to the application layer. This way, the application layer is not burdened with low-level detection tasks, but is responsible for fault management activities (e.g., alarm logging, base-station notification, fault recovery techniques). REDFLAG does not assume any specific Routing and MAC protocols so that maximum control of packet delivery, duty-cycling, etc. is still available to the developer. Beyond layer independence, REDFLAG has been designed to be effective in a variety of settings. It performs well for different network topologies and node capabilities, without requiring any resource-plentiful motes. It is capable of operating in both event-driven and periodic data collection. It provides low-cost, distributed fault detection for dense networks, but it is also suitable in sparse networks where individual node health is more of a concern. It can be deployed in a network prone to significant fault rates just as easily as in one which is inherently stable, without introducing prohibitive resource costs.

II. REDFLAG SERVICE DESCRIPTION

REDFLAG is a distributed fault detection and diagnostic service for sensor applications. It provides two subservices: the Sensor Reading Validity subservice (SRV) and the Network Status Report subservice (NSR). The former applies signal processing and rule-based techniques to validate sensor readings, so that abnormal readings are recognized. The latter collaboratively monitors WSN nodes and link connectivity to abate data loss. Subservices provide clean interfaces for both service configuration and report notification purposes. Details of these subservices now follow.

A. Sensor Reading Validity Subservice

WSN applications are designed to monitor all kinds of phenomena using diverse sensor types. Sensors translate a physical magnitude of interest into human or machine readable signals. However, the translation process is subject to many non-ideal factors: sensitivity variations, scale and offset dynamic errors, calibration drifts, hysteresis, noise, non-linearity effects, etc. Systematic errors (influenced by offset, scale ranges, sensitivity variations, non-linearity, etc.) may be handled by calibration; whereas, random errors (primarily noise) may be compensated for with signal processing techniques.

Calibration is the process of determining the relationship between the output signal and the input magnitude in a sensor. A linear relationship is characteristic of many sensors, i.e., output = \( \alpha \cdot \text{input} + \beta \). Although sensors may come pre-calibrated, the calibration may need adjustment, as it is influenced by many factors (including remaining power, surrounding temperature and humidity, and deployment lifetime).

As an electronic component, a sensor’s output signal is subject to noise; which is commonly characterized by its stochastic properties. The reading given by a sensor is related to the physical magnitude of interest and distorted by noise. Signal processing is the method used to recover real values from a noisy set of readings.

The SRV service provides an interface to access validated sensor readings. For each ‘read’ command triggered by the application layer, SRV returns both the recovered physical value and its validity indicator. The validity field denotes whether the reading is valid (i.e., it passes the applied rules) or, if it is invalid, which type of failure(s) was detected. Validation is performed based on the following algorithm.

**Overview of the SRV Algorithm**: The basis of the SRV subservice is to (in)validate sensor readings using a set of rules. First, basic signal processing is applied to sampled sensor outputs in order to minimize the impact of noise on provided readings (i.e., avoid random errors). Assuming that noise, \( \eta \), (modeled as Additive Gaussian White Noise: \( \bar{\eta} = 0 \)) and the magnitude of interest, \( \upsilon \), are independent variables and that the real magnitude is stable in a given sampling interval (\( \sigma_\upsilon \approx 0 \)), SRV computes the mean and the standard deviation of \( N \) rapid, contiguous sensor outputs:

\[
\bar{R} = \frac{1}{N} \sum_{i=1}^{N} R_i = \frac{1}{N} \sum_{i=1}^{N} (\upsilon_i + \eta_i) = \frac{1}{N} \sum_{i=1}^{N} \upsilon_i \approx \upsilon,
\]

\[
\sigma_R^2 = \frac{1}{N} \sum_{i=1}^{n} \sigma_{(\upsilon + \eta)_i}^2 = \frac{1}{N} \sum_{i=1}^{n} (\sigma_{\upsilon_i}^2 + \sigma_{\eta_i}^2) = \frac{1}{N} \sum_{i=1}^{n} \sigma_{\eta_i}^2 \approx \sigma_\eta^2
\]

Therefore, a meaningful sensor reading value, \( \bar{R} \), and the noise’s influence on that reading, \( \sigma_R^2 \), are available for analysis. Based on faults classified from previous sensor deployments (cf. [4], [7] and [9]), the following practical set of rules are defined:

- **Noisy Reading**: The reading is undesirably noisy. That is, the reading’s standard deviation exceeds the expected maximum noise threshold: \( \sigma_R^2 > \sigma^2_{\text{max}} \).
- **NLDR Reading**: The sensor value may fall outside the range of calibration (i.e., the Linear Detection Range (LDR)), which is bounded by \( \text{LLB} \) (Linear Lower Bound) and \( \text{LUB} \) (Linear Upper Bound): \( \bar{R} > \text{LUB} \) or \( \bar{R} < \text{LLB} \).
- **Out of Range Reading**: The reading is completely nonsensical if it does not fall inside the total detection range of the sensor [\( \text{TLB}, \text{TUB} \)] (Total Lower Bound, Total Upper Bound): \( \bar{R} > \text{TUB} \) or \( \bar{R} < \text{TLB} \).
- **Stuck Reading**: Minimal instability is expected in a set of rapid, contiguous sensor readings, i.e., \( \sigma_R^2 > 0 \). Therefore, an unusually steady set of readings may be indicative of sensor failure (e.g., see STUCK in [1]), that is: \( \sigma_R^2 < \sigma^2_{\text{min}} \).
- **Abruptly Changed Reading**: Due to erratic hardware or an environmental event, it may be the case that a reading is anomalously different from the previous one. An application dependent threshold for the rate of expected change is used: \( \frac{|R_t - R_{t-1}|}{\Delta t} > \Delta_{\text{max}} \).
By applying the rules above, both systematic and random sensor errors are detected (then, reported) by the SRV sub-service, thus reducing data uncertainty.

**SRV Service Parameter Determination:** SRV is designed to be adaptive to dynamic WSN conditions, such as sensor calibration shifts or changing surrounding conditions. Using available SRV configuration commands, all of the following service parameters may be adjusted at will by the application:

- $\sigma^2_{\text{min}}$: $\sigma^2_{\text{min}}$ reflects the minimum expected variability on sensor readings. An estimation may be obtained by analyzing measurements in controlled laboratory conditions, setting it to be less than or equal to the minimum observed variability.
- $\sigma^2_{\text{max}}$: Due to noise influence, contiguous sensor readings might show some variability. $\sigma^2_{\text{max}}$ may be determined empirically by taking multiple readings while holding the variable of interest constant and calculating the standard deviation. Sensor accuracy, reported in the sensor documentation, should also be taken into account.
- **TLB** and **TUB**: Specific sensor documentation often declares the extreme lower and upper bounds of the sensor’s working range. In practice, the application may utilize more stringent bounds to ensure a tighter data range.
- **LLB** and **LUB**: The linear lower and upper bounds should be determined empirically using in-situ or ex-situ calibration methods appropriate for the given sensor. Note that if a non-linear calibration model is employed, these parameters should still be used to bound the valid calibration range.
- $\Delta_{\max}$: Typically there is some expectation of continuity between successive readings (e.g., see the SHORT rule given in [1]). For a series of consecutive sensor values, $R_{i+1}$ and $R_i$, over a period of $t_i$ time units, one may compute: $\Delta_E = \max_i E\left[\frac{R_{i+1} - R_i}{t_i}\right]$. $\Delta_{\max}$ should then be chosen such that $\Delta_{\max} \geq \Delta_E$.
- $\alpha$ and $\beta$: The calibration parameters can be adjusted for each sensor at any time. The impact of sensor calibration drift on data can be minimized by carefully adjusting $\alpha$ and $\beta$ parameters as it occurs.

**B. Network Status Report Subservice**

Typical WSN deployments are subject to unforgiving environments where node and link failures are likely to happen. Nodes may appear unresponsive if they run out of battery, are broken, or if connectivity with their neighbors is lost. Missing data from unresponsive nodes is a serious problem. Moreover, specific applications, e.g. [3], may require ancillary information such as notification of temporary link failures. Hence, the primary goal of the NSR subservice is to detect unresponsive nodes and diagnose their root causes.

Lowering false alarm rates is the primary concern of this fault detection and diagnostic service. Hsin and Liu [10] presented and validated an efficient two-phase fault detection approach. In the first phase, each node monitors its neighbors and, in the second, it corroborates its findings with other neighbors before sending an alarm. However, the algorithm has several drawbacks that limit its applicability. First, it uses fixed timers for both phases which forces unnecessary resource usage (radio, CPU). Furthermore, these timers are derived based on unrealistic assumptions, namely that transmissions in nodes are independent, and that the transmission of a node’s neighbor follows a Poisson distribution. Second, a neighbor is locally declared dead if a single packet is missed, and only one message sent by each neighbor is used to reject a local alarm. Due to the failure prone nature of sensor networks, this mechanism is problematic and decreases fault detection accuracy. Third, alarms are always sent to the base station at the end of each fault detection period. This could cause unnecessary traffic in WSN scenarios where applications are not interested in frequent fault detection reports.

NSR leverages the insights gained from [10] and uses a similar two-state approach, i.e., confirmation of local decisions by adopting neighborhood agreements. Although, at a higher level NSR bears certain similarity to [10], the detailed algorithm is remarkably different. NSR aims to be flexible enough to accommodate different sensor deployments and diverse application requirements, such as the need for a more stringent detection latency, or the ability to adjust timing periods for duty cycling. NSR also has the ability to estimate the cause of failures, which is useful for WSN management. This is achieved by (1) providing clean interfaces to the application and communication layers so they can easily interact with the fault detection algorithm; and (2) adopting a tunable timing scheme to make the NSR service applicable to many scenarios and applications. We elaborate on these ideas in the following sections.

![Fig. 2. Timeline of the NSR algorithm](image)

**Overview of the NSR Algorithm:** The algorithm operates using a tunable timing schedule\(^2\), depicted in Fig. 2. $T_d$ (detection epoch) indicates how often a node starts detection activities. At the beginning of each detection epoch, a node turns on its radio (if not already started) and communicates with its neighbors. This neighborhood communication process continues for $T_n$ (neighboring epoch), then the node can shut off its radio and processor for the remaining time ($T_s$) before the next detection epoch. Epoch durations can be customized to meet specific detection latency and duty cycling requirements.

\(^2\)Clock synchronization among nodes is assumed, which can be achieved using an existing algorithm, cf. [11].

---

1 In this paper only linear calibration is considered, but REDFLAG may be easily modified to work with any calibration model.
The algorithm’s logic is split into two distinct states in order to decrease false alarm rates: the Local Detection (LD) state and the Neighbor Consensus (NC) state. While in the LD state, each node, \(n_i\), monitors each neighbor, \(n_j\), by listening to \(n_j\)’s messages. To accommodate transient failures, only if \(k_{1}^{\text{max}}\) messages are missed from a particular neighbor, will that neighbor be considered suspicious. Once this happens, node \(n_i\) transitions to the NC state concerning \(n_j\). In that state, node \(n_i\) exchanges information with its neighbors for a maximum of \(k_{2}^{\text{max}}\) epochs to consensually determine the status of the suspicious node. Verdicts of unresponsive nodes are reported to the application layer when \(T_r\) fires.

Details of the NSR Algorithm: The key data structure that each node maintains is the neighbor table (NT), where all the needed information about its neighbors is stored and updated. That is, for its \(j^{th}\) neighbor, denoted as \(n_j\), node \(i\) (\(n_i\)) stores: neighbor’s unique ID, counters \(k_{1}^{i}[j]\) and \(k_{2}^{i}[j]\), the remaining energy in neighbor \(j\), and the link quality between \(n_i\) and \(n_j\). All this neighborhood information is continuously updated within each \(T_n\) epoch, which makes the algorithm flexible enough to handle dynamic joining and leaving of nodes.

A node is required to keep a minimum of \(NS_{\text{min}}\) and a maximum of \(NS_{\text{max}}\) number of nodes in its neighbor table. Because of mote memory limitations, a mote should not monitor too many nodes. At the same time a neighbor table should not be too small in order to avoid orphan nodes. Therefore, neighbor inclusion policy must be dynamic. Taking an arbitrary node, \(n_i\), another mote is considered \(n_i\)’s \(j^{th}\) neighbor if the link quality, \(\Lambda_j^{i}\), is greater than \(\Lambda_{\text{min}}\) (the Link Quality Threshold). \(\Lambda_{\text{min}}\), nevertheless, is adjusted each \(T_n\) depending on the actual NT size: lowered when having fewer neighbors than required, raised when having too many neighbors.

Four types of messages are exchanged in each node’s one-hop neighborhood to monitor and discuss neighbor status; these are described in Table I.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>HELLO ((n_i))</td>
<td>hello</td>
<td>broadcast by node (n_i) to notify surrounding nodes it is alive</td>
</tr>
<tr>
<td>ALQ ((n_i, n_j))</td>
<td>alarm query</td>
<td>broadcast by node (n_i) to inquire about the status of node (n_j)</td>
</tr>
<tr>
<td>ALR ((n_i, n_j, k_1))</td>
<td>alarm rejection</td>
<td>broadcast by node (n_i) to indicate that doubtful node (n_j) is still alive (remaining (k_1) included)</td>
</tr>
<tr>
<td>ACK ((n_i, n_j, n_k))</td>
<td>alarm rejection ack.</td>
<td>unicast by node (n_i) to node (n_j) to acknowledge the reception of (n_j)’s ALR about doubtful node (n_k)</td>
</tr>
</tbody>
</table>

In each neighboring epoch \((T_n)\), node \(n_i\) updates the status of each node in its NT by processing the messages it receives. It may then transmit alarm queries, acknowledgments and alarm rejection messages. Fig. 3 depicts the logic a node, \(n_i\), follows for each neighbor, \(n_j\), in its NT. Note that alarm query messages are expected to be answered in the next neighboring epoch. In the event that no message needs to be sent, \(n_i\) broadcasts a HELLO to assure its neighbors that it is still alive. NSR algorithm’s finite state machine (Fig. 3) is now described in detail:

1) Initial State: As previously mentioned, node \(n_i\) will add node \(n_j\) to its NT, then denoted as \(n_j\), if \(\Lambda_j^{i}\) is greater than \(\Lambda_{\text{min}}\). Once this occurs, \(n_i\) enters into the FSM for node \(n_j\), starting in the Local Detection state. Note that \(n_i\) is involved in different states for each neighbor.

2) Local Detection State: In this state, \(n_i\) considers \(n_j\) to be alive. \(k_{1}^{i}[j]\) counts down (from \(k_{1}^{\text{max}}\) to 0) the number of epochs that \(n_i\) has not heard from \(n_j\). Whenever \(n_i\) hears from \(n_j\), \(k_{1}^{i}[j]\) is reset. If node \(n_i\) receives an ALR about node \(n_j\), \(k_{1}^{i}[j]\) is updated with the larger value between its current value, \(k_{1}^{i}[j]\), and the value received in the message (say, from neighbor \(n_k\)). Updating \(k_{1}^{i}[j]\), instead of resetting it, reduces detection latency and avoids potential loops. While in this state, \(n_i\) rejects all received ALQ messages about \(n_j\) by sending an ALR response. Finally, if \(n_i\) does not hear from \(n_j\) in \(k_{1}^{\text{max}}\) epochs, then it transitions to the Neighbor Consensus state.

3) Neighbor Consensus State: In each neighboring epoch within this logical state, node \(n_i\) broadcasts an alarm query (ALQ) to its neighbors in regards to \(n_j\). It then waits for a maximum of \(k_{2}^{\text{max}}\) epochs to receive an ALR about \(n_j\) before deciding that the node is unresponsive and transitioning to the Decision Made state. The number of epochs waited are counted down using \(k_{2}^{i}[j]\). Upon obtaining an ALR from, say, node \(n_k\), node \(n_i\) transitions back to the Local Detection state and adopts \(k_{2}^{i}[j]\) as its own. It also acknowledges that it has received this information by transmitting an ACK to \(n_k\). Note that, while in this state, \(n_i\) does not respond to ALQs about \(n_j\) since it is also in doubt about \(n_j\)’s status.

4) Decision Made State: The application is notified about an unresponsive node at the end of each reporting period, \(T_r\). The node remains in this state until it is time to report the detected failure. However, if \(n_j\)’s recovery is discovered by this node before notifying the alarm, \(n_i\) cancels the report and accordingly returns to Local Detection (avoiding false alarms). Lastly, when a report is delivered, \(n_i\) terminates monitoring \(n_j\) by removing it from its NT.

This logic is followed by a node \(n_i\) for each neighbor in its NT (\(n_j\)), having multiple FSM running simultaneously within each neighboring epoch \((T_n)\). What actions need to be taken and what messages need to be sent within \(T_n\) is therefore determined by the combination of multiple FSM states.

Network Status Reports: The ultimate goal of NSR is to provide the application accurate picture of the general network health. To keep NSR flexible, different report types
and timings are available. On one hand, the application layer chooses when to receive detected failure notifications by adjusting $T_a$; on the other, it may also select all or some of three different reports: (1) the Local Detection report, based on temporary decisions made within the LD state (i.e., information about temporary failures); (2) the Neighbor Consensus report, reflecting decisions made after completing the LD state (i.e., which nodes are tagged as suspicious); and (3) the Final Decision report, notifying decisions made after going through both the LD and NC states.

Nevertheless, the network status report format is standard in all cases. It includes faulty node’s ID, report type, fault severity (in the form of $k_{i \rightarrow j}/k_{i \rightarrow j}^{\text{max}}$ and $k_{i \rightarrow j}^{\text{max}}/k_{i \rightarrow j}^{\text{max}}$), and the estimated root cause of failure. Root causes detected by the presented service include: a node running out of battery, a link failure and an unexpected failure. Distinction among them is based on remaining energy information and link quality indicators collected within each neighboring epoch. If the remaining energy of a neighbor is below the Low Battery Threshold, then low battery is the reported root cause. Similarly, links to neighbors that fall below the Link Quality Threshold are tagged as link failures. When the cause of unresponsiveness cannot conclusively be narrowed down, the unexpected label is issued.

NSR Service Parameter Determination: The correct performance of the NSR subservice for a given network topology and deployment depends on the optimal selection of the following parameters: $N_{s_{\text{min}}}, N_{s_{\text{max}}}$ and $\Lambda_{\text{min}}$. The values for these parameters are either based on theoretical analysis (distance between nodes, used radio device features, etc.) or empirical data for the current or a previous deployment. The selected values should ensure each node in the network is monitored by at least one other mote running NSR. Note that application-specific $\Lambda_{\text{min}}$ is used as an initial estimation, but will then be dynamically adjusted on a node by node basis.

$k_1^{\text{max}}, k_2^{\text{max}}, T_d$ and $T_n$: Although the algorithm logic stays the same, the adjustment or selection of the timers renders NSR useful to a variety of applications. The application determines when to run detection activities by specifying $T_d$. The detection latency of the NSR algorithm is $(k_1^{\text{max}} + k_2^{\text{max}})T_d$. Therefore, in order to detect nodes which have been unresponsive for $T_f$ time units, $T_d \geq (k_1^{\text{max}} + k_2^{\text{max}})T_d$ must be fulfilled. It is recommended to avoid usage of small $k_1^{\text{max}}$ and $k_2^{\text{max}}$ values; otherwise, temporary failures may be reported as permanent. $T_n$ is dependent on the maximum latency of the MAC layer being utilized, $T_{\text{MAC}}$, and the neighborhood size: $T_n \geq T_{\text{MAC}} \times N_{s_{\text{max}}}$. This way, enough time to communicate (in the worst case) with each neighbor is assured. Detection related activities, including message exchanges in the neighborhood and decision makings, are all conducted within each neighboring epoch. At the end of each $T_n$, NSR notifies the application about the termination of the current round of detection, implying that the next round will not start until the next $T_d$ and the mote can be put to sleep until then. Sleeping period $T_s$ is then computed based on $T_d = T_s + T_n$.

NSR Reporting Period ($T_r$): This parameter should be set according to the application’s needs, so that the application layer has the flexibility to decide when and how to manage provided alarm reports. It can log them, or send them to the base station via wireless links or serial interfaces. For instance, fault sensitive applications may wish to receive NSR reports as soon as the faults are detected ($T_r = T_d$), while other applications may just be interested in periodic information matching its data collection schedule ($T_r \geq T_d$). Note that the reporting period is bounded by the detection epoch and desired detectable failure duration: $T_d \leq T_r \leq T_f$. By careful selection of $T_r$, the application layer is able to separate or mix
data and alarm report traffic loads. The only restriction is that $T_r \geq T_d \geq T_n$, since it takes $T_n$ to collect information about the network.

In summary, the application has the full control of the NSR subservice. Careful determination of the NSR parameter set allows the application layer to meet specific scenario requirements: detection latency, report latency, report frequency, report types, duty cycles of nodes, data collection time, etc.

### III. PERFORMANCE EVALUATION

The REDFLAG fault detection service has been implemented in TinyOS 2.x [12], an open-source operating system specifically designed for resource-constrained wireless sensors. We have decided to use TinyOS due to its acceptance in the WSN community and its portability to many hardware platforms. The design of TinyOS also matches perfectly with REDFLAG’s goal of supporting many sensor applications, and the layered architecture of REDFLAG can be easily supported by the components and interfaces in TinyOS’s programming structure. Applications using REDFLAG may access the SRV and NSR subservices independently. Usage of REDFLAG is simplified to (1) parameter configuration, (2) initialization, and (3) report analysis.

In order to evaluate REDFLAG in a variety of WSN environments, its performance has been studied using TOSSIM [13] – a TinyOS simulation tool which realistically reproduces WSN physical and link layer features [14]. Simulations rather than real-world empirical studies are used because (1) it is prohibitively expensive to deploy real WSNs for different scenarios that can validate and demonstrate the effectiveness of REDFLAG; and (2) both abnormal sensor readings and network faults are much easier to reproduce in simulation than in a real deployment.

To accompany fault detection accuracy, energy consumption is also used to evaluate REDFLAG’s performance. Since radio communication consumes two to three orders of magnitude more power than computations [15], only radio communication cost was considered in the simulations. The remaining energy in each node was calculated in real-time by computing radio transmission, reception, and idling periods, using power consumption values from the CC2420 radio [16].

#### A. Simulation Setup

To add to TOSSIM’s realism, a variety of sensor and unresponsive node faults were created as follows.

**Sensor Failure Creation:** Systematic and random sensor errors are both simulated. The justification for the fault creation methods comes from the analysis by Ramanathan et al. [7] and by Porta [4]. In [7], post-experimental analysis suggested that almost 40% of the approximately 25,000 data values were faulty. Of the received data points, 1% were excessively noisy, 11% were outside the sensor’s calibration range, 12% were contiguous points outside the total detection range, and 26% had erratic “shorts” where the readings would drop to zero for a short time and then resume normal operation. Porta [4] deployed sensors in an experimental sand test bed over a period of several months. Ion chromatograph analysis of manual aqueous samples indicated a drastic drift in calibration, relating electric conductivity to sodium bromide concentration, after only one month of deployment. Her data was also subject to significant random noise.

Based on the above studies, each sensor reading is systematically perturbed in simulation by: (1) adding random gaussian noise; and (2) slightly drifting the calibration. Moreover, Bernoulli Processes were used to generate the following random failures:

1) **Stuck readings (0.1% chance of occurrence):** the sensor continues to report the approximately same value;
2) **Out of range (0.1% chance):** the sensor will report a value out of range for a contiguous period of time;
3) **Abrupt shifts (1% chance):** the linear calibration curve will shift randomly;
4) **Noisy reading (1% chance):** the variance in random noise for each reading is increased.

Realistic sensor reading values were obtained from subsurface contaminant plume experiments [4]. Concentration data was constructed by using finite difference models for the governing partial differential equations based on this experimental design. Then, systematic and random errors were introduced, simulating real sensor operation.

**Unresponsive Node Failure Creation:** WSN deployment conditions are accurately simulated by TOSSIM 2.x [14]. It uses a closest-fit pattern matching noise model, a SNR-based (Signal to Noise Ratio) packet error model with SNR-based interference and CSMA. Therefore, it simulates the hidden terminal problem, the exposed terminal problem, strongest-first versus stronger-second, and other wireless communication issues. These provide a realistic sensor network environment where dynamic and temporary link failures are likely to happen. However, in order to evaluate REDFLAG more accurately, nodes running out of battery, unexpected sudden mote failures, and disconnected nodes have been artificially added.

In real deployments, physical obstructions (human constructions, trees, etc.) and radio device failures (e.g., hardware problems) may cause nodes to appear completely disconnected even if they are still collecting and processing data. To simulate these phenomena, randomly selected nodes were disconnected at a random time governed by a Bernoulli Process. Disconnection is simulated by deleting all links to the rest of the network in TOSSIM’s radio model. Disconnected nodes were again reconnected after a specified time interval. Additionally, we simulated unknown random node failures. These could also recover from failure. In TOSSIM, nodes that reached a minimum energy level (based on continuous power consumption monitoring) were halted in order to simulate insufficient mote battery power.

Failures in WSNs may appear in an isolated or patterned manner. To simulate the first, faulty nodes were selected based on a Bernoulli process. For the second, the center and the radius of the faulty area were specified first, then all nodes within that area were disconnected or made unresponsive.
B. Performance Results

In the following results, when referring to a grid of size \( x \) we mean a grid containing \( x \) nodes each 10m apart. When showing different topologies/densities, we indicate that \( x \) number of nodes are placed in a 40m \( \times \) 40m region according to that specific topology.

**REDFLAG is lightweight**: REDFLAG is claimed to be lightweight, both in memory resources and energy consumption. Its required memory footprint for this TinyOS implementation is (for different platforms): 26660 bytes in ROM and 2214 bytes in RAM (micaz), 25824 bytes in ROM and 2362 bytes in RAM (telosB), 25778 bytes in ROM and 2235 bytes in RAM (tinynode).

Results shown in Figure 4 demonstrate that REDFLAG’s average power consumption (with both subservices simultaneously working) is minimal and scales well for different network sizes and densities. If we assume motes are powered by two AA alkaline batteries (18000J) and take the most energy consuming case (500mJ in 8 minutes) from the set of results, the network lifetime using REDFLAG is 200 days.

**Performance of the Sensor Reading Validity Subservice**: Using initial parameter values based on data from [4], the average percentage of detection for each type of fault is given by the light-colored bars in Figure 5.\(^3\) While “out of range” ([\( TLB, TUB \]) = [0, 5000] and [\( LLB, LUB \]) = [27, 128]) faults are easily detected, others are sometimes missed (note that “out of range” includes all faults outside linear and total detection ranges).

To improve this, the three parameters, \( \sigma_{\text{min}}, \sigma_{\text{max}}, \) and \( \Delta_{\text{max}} \) were adjusted to more conservative, but reasonable, values. Detection percentages of abnormal readings improved dramatically, as indicated by the dark-colored bars in the figure.

Deciphering between an interesting and an erroneous data value is difficult in an actual deployment. SRV parameters might need to be dynamically adjusted to increase detection accuracy. We contend that this decision is best left up to the application; for instance, by using predictions from a model (e.g., [6]). Again, REDFLAG is providing a detection service to the application layer, which is in the best position to update the SRV parameter set. Here, it has been demonstrated that, once appropriate parameters are chosen, SRV is capable of detecting most of the common errors highlighted by previous research.

![Fig. 4. REDFLAG energy consumption](image)

![Fig. 5. SRV Performance](image)

**Performance of the Network Status Report Subservice**: We first demonstrate the parameter selection heuristics. Using these parameters, the NSR subservice is then evaluated for a variety of scenarios, in terms of different network types and failure patterns.

Minimum and maximum neighborhood size (\( NS_{\text{min}} \) and \( NS_{\text{max}} \)) and the link quality threshold values were chosen to be 4, 6, and -85 dBm, respectively. These were found by analyzing the distance between nodes, the radio capabilities and the noise trace used in this simulation set; assuring that each node monitors at least all of its one-hop neighbors. \( T_n \) is related to the maximum delay of the MAC layer (\( \tau_{\text{MAC}} \) = 10ms in this case), that is: \( T_n \geq \tau_{\text{MAC}} \cdot NS_{\text{max}} \); \( T_n \) = 100ms. In order to let the nodes go to sleep, \( T_d \) was fixed to be greater than \( T_n \); here, \( T_d \) = 1000ms (\( T_s \) = 900ms). The reporting period, \( T_r \), was selected to be the same as the detection period, \( T_r = T_d \), so that failures are reported as soon as they are detected.

\( k_{\text{max}}^1 \) and \( k_{\text{max}}^2 \) remain to be selected. From Figure 6, it can be seen that the optimal detection accuracy is obtained when \( T_f \geq (K_1 + K_2) T_d \) (the lower left triangle of the graphs in the first row) for any network size. Based on the second row, the energy consumption is generally smaller if \( k_{\text{max}}^2 \geq k_{\text{max}}^1 \) (the left upper triangle). Lastly, from the final row, larger values of \( k_{\text{max}}^1 \) and \( k_{\text{max}}^2 \) yield fewer reports. This analysis leads to the selection of \( k_{\text{max}}^1 = 4 \) and \( k_{\text{max}}^2 = 6 \).

Using these parameter values, detection and diagnosis accuracy of the presented NSR algorithm is evaluated in different scenarios. In Figure 7, mean detection accuracy for isolated faults is shown to be always above 90% for different network sizes, network densities, failure durations, and failure probabilities. The grey area in each graph denotes a worst case 95% confidence interval of 5 runs assuming a t-distribution.\(^4\)

\(^3\)The standard error from the student t-distribution were all less than 0.01% for the 60 simulations, so they were omitted from the plot. We also analyzed the results for false alarms and did not find any.

\(^4\)The computed standard error would be much smaller if more simulations were performed, but this was not undertaken due to limited computational resources.
Figure 6 depicts how NSR performs in a 100-node grid in the presence of patterned failures. When having small faulty sensor groups, NSR provides high detection accuracy but, as the faulty area increases, detection accuracy decreases. This is because live nodes can only monitor the perimeter of the faulty area, not nodes in the middle.

Figure 8 shows the diagnosis accuracy in a 25-node grid topology. NSR succeeds in recognizing energy depletion within the network and acceptably diagnoses link failures. Diagnosing the root cause of an unexpected failure remains difficult; still, NSR reliably detects the failures. We hypothesize that the availability of better link quality indicators will improve diagnosis accuracy – REDFLAG currently uses the received packet strength in TOSSIM.

The above results are based on Final Decision Reports, which report permanent failures that may require manual intervention. Recall that the NSR subservice also provides intermediate reports, named Local Detection Reports and Neighbor Consensus Reports. Availability of this kind of information is justified in scenarios where knowledge of temporary packet losses at exact times is desirable. The application may then track the severity of network misperformance, possibly justify missing information from some specific nodes/areas, and recover from those situations.

IV. CONCLUSION

REDFLAG, a Run-timeE, Distributed, Flexible, Lightweight And Generic fault detection service, has been described by detailing its SRV and NSR subservices. Without making any network assumptions, the design of REDFLAG as an independent layer facilitates its integration with a myriad of applications. Furthermore, REDFLAG's online adjustability provides run-time reconfiguration capabilities by using configure commands through TinyOS interfaces.

Failure detection accuracy of both SRV and NSR are shown to be highly satisfactory. Optimal service parameter determi-
nation methodologies were presented in order to match each particular application’s requirements. Performance evaluation also demonstrates how REDFLAG is flexible and lightweight with negligible impact on detection accuracy.

NSR accurately diagnoses low power and link failure root causes. This advocates the conclusion that NSR diagnoses correctly when good indicators are available. Future work may focus on adopting other link quality estimation metrics (as proposed in [17]) and additional node status indicators (e.g., debugging messages, reboots, illegal memory writings, and stack status) to further improve diagnosis accuracy.

Future work will also explore the collaboration of REDFLAG with specific applications, such as in [3]. We hope that this approach to fault detection will prove useful in many other real WSN applications.

REFERENCES


