**REDFLAG: A Run-timE, Distributed, Flexible, Lightweight, And Generic fault detection service for data-driven wireless sensor applications**

Iñigo Urteaga\(^{a}\), Kevin Barnhart\(^{b}\), Qi Han\(^{a,\ast}\)

\(^{a}\) Department of Math and Computer Sciences, Colorado School of Mines, Golden, CO, United States
\(^{b}\) Division of Environmental Science and Engineering, Colorado School of Mines, Golden, CO, United States

**ABSTRACT**

Increased interest in wireless sensor networks by scientists and engineers is forcing wireless sensor networking 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. However, 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 wireless sensor 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. Being unrestrictive, REDFLAG is generically available to a myriad of applications and scenarios. As a matter of fact, REDFLAG has been applied into a subsurface contaminant transport model to improve the model performance in the presence of erroneous sensor data.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Wireless sensor networks (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 the 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., [3–5]). 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) sub-service, which detects erroneous sensor readings, and a Network Status Report (NSR) sub-service, 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 unrestricted, it is generally available to a myriad of applications and scenarios.

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 nodes. 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. To the best of our knowledge, this is the first paper that addresses in a unified framework both abnormal and missing sensor readings, the two main challenges in data-driven sensor applications.

As an example, we next briefly describe the subsurface contaminant plume monitoring application and how it interacts with REDFLAG. Toxic chemicals and biological agents are released into the subsurface as a result of accidental spills, improper disposal, or intentional damage. These releases cause migrating plumes in the subsurface with concentrations that are spatially distributed and transient, posing potential risk to humans and the ecological environment. Using wireless sensor networks, hydrological data may be collected in situ without manual sampling and analysis [1,4,6,7]. These data may then be used by computational models of hydrologic processes (often running at resource sufficient servers) for identification of plume parameters and prediction of plume behavior. However, sensor networks are failure prone, leading to non-negligible amount of anomalous data. To improve the performance of these computational models, REDFLAG may be utilized to reduce sensor data uncertainty as follows. Data collected from each sensor will first be passed through REDFLAG at each node. REDFLAG provides an indicator of data fault types along with the data before they are transmitted by the application layer to the central server. The server then processes REDFLAG’s messages and generates filtered data set for the computational models. For instance, if a sensor reading is flagged as faulty, that data will not be included in the filtered data set. More detailed discussion on using REDFLAG for this application may be found in Section 5.

The rest of this paper is organized as follows. Section 2 discusses existing work that is most relevant to ours. Section 3 presents the detailed algorithm for the two subservices of REDFLAG: one tackles abnormal sensor readings, and the other addresses missing sensor readings. To improve REDFLAG’s usability, practical guidelines are provided on how to select various service parameters. Extensive performance studies have been conducted to evaluate the performance of REDFLAG under various system and network conditions, and we report the results in Section 4. Further, we have incorporated REDFLAG to a specific application where the performance of groundwater contaminant transport models is studied in the presence of various faults inherent in wireless sensor networks. Results presented in Section 5 indicate that REDFLAG has significantly improved application’s performance by the reduction of data uncertainty. We conclude the paper with a few suggested areas as future work in Section 6.

2. Related work

The work related to REDFLAG can be found in three sub-areas: fault detection systems, abnormal data detection techniques, and node fault detection (leading to missing data) techniques. We next discuss each of them in more detail.

1) Fault detection systems. REDFLAG is distinctly different from previously proposed WSN fault detection schemes. Studies like [2] only consider query-driven data collection scenarios. Other systems (e.g., Sympathy [8], SNMS [2]) are based on centralized approaches, where diagnosis data is periodically reported to a centralized server. Additional alternatives (e.g., [9]) address various networking faults in isolation, such as power depletion, hidden terminals, congestion detection and mitigation, and asymmetric communication links. In contrast, REDFLAG is the first to reduce data uncertainty by detecting
and identifying the root causes of the two most worrisome issues in data-intensive wireless sensor applications: abnormal and missing sensor readings. In addition, the layered design and unrestrictive nature of REDFLAG allows its integration with many existing fault detection solutions.

(2) **Abnormal data collection techniques.** Work in this area differs in their assumptions of the observed phenomena, which then determine the techniques to identify anomalous data. Some expect the phenomena to follow relatively simple models (e.g., [10]), while others require a dense network to ensure spatio-temporal relationships in the data [11–15]. For instance, spatio-temporal correlations and sample redundancies are used to validate the data faults identified at local tiers [12], or the correlations between different attributes as well as spatio-temporal correlations are first captured by probabilistic data models and then used to tolerate data loss and noise [13,14]. In addition, non-parametric and unsupervised methods based on data exchanges among neighbors have been developed for outlier detection [16], a kernel density estimator has been used to estimate data distribution and identify distance- or density-based outliers in a distributed manner [17]. We note that REDFLAG can incorporate the outlier detection approaches summarized above for applications that possess cross-attribute correlations or spatio-temporal correlations. REDFLAG relies on simple rule-based methods to detect outliers without incurring any message exchanges with neighbors. Further, it provides both raw sensor readings and validity indicators to the application, instead of concealing outliers using in-network aggregation which may not be needed by the application.

(3) **Node fault detection techniques.** Node fault detection has been previously addressed via a neighbor consensus process as in [18] and Memento [19]. In [18], a two-phase solution, consisting of a monitoring phase and an alarm rejection phase, is used in order to reduce false alarm rates. The duration of each phase is probabilistically computed based on a simplistic link model using the expected number of collisions for the network traffic rate. Memento [19] has the similar two phases as [18]. Nevertheless, it is based on the mean and standard deviation of the number of consecutively missing heartbeats that are typical of each live neighbor and user-specified tolerable false rate. This way, it estimates a bound on the maximum number of heartbeats missed before declaring the node’s death. Although REDFLAG’s node/link failure detection algorithm also uses two-phase logic, it is not confined to a specific link model and it identifies the cause of a broken connection. REDFLAG further provides a flexible node failure detection timing scheme suitable for various application needs.

3. **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. The design choice of having two subservices working independently from each other was motivated by our collaborative work with experts from application domains (e.g., [3–5]). The fact that the two subservices can be separately activated or deactivated makes REDFLAG unrestrictive and generically available to a myriad of applications. Details of these subservices now follow.

3.1. **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., $output = \alpha \cdot input + \beta$. 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 thus 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.

**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, $\nu$, are independent variables and that the real magnitude is stable in a given sampling interval ($\sigma^2_\nu \approx 0$), SRV computes the mean and the standard deviation of $N$ rapid, contiguous sensor outputs:

$$
\bar{R} = \frac{1}{N} \sum_{i=1}^{N} R_i = \frac{1}{N} \sum_{i=1}^{N} (\bar{\nu}_i + \bar{\eta}_i) = \frac{1}{N} \sum_{i=1}^{N} \bar{\nu}_i \approx \bar{\nu},
$$

$$
\sigma^2_R = \frac{1}{N} \sum_{i=1}^{N} \sigma^2_{\nu_i + \eta_i} = \frac{1}{N} \sum_{i=1}^{N} (\sigma^2_{\nu_i} + \sigma^2_{\eta_i}) = \frac{1}{N} \sum_{i=1}^{N} \sigma^2_{\eta_i} \approx \sigma^2_{\eta},
$$

\[ \]
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 [4,8,20], 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: $\sigma_R^2 > \sigma_{max}^2$.

- **NLDR Reading**: The sensor value may fall outside the range of calibration, the Linear Detection Range (LDR) bounded by $LLB$ (Linear Lower Bound) and $LUB$ (Linear Upper Bound): $\bar{R} > LUB$ or $\bar{R} < LLB$.

- **Out of Range Reading**: The reading is completely non-sensical if it does not fall inside the total detection range of the sensor [$TLB$, $TUB$] (Total Lower Bound, Total Upper Bound): $\bar{R} > TUB$ or $\bar{R} < TLB$.

- **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: $\sigma_R^2 < \sigma_{min}^2$.

- **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: $\frac{|\bar{R}_i - \bar{R}_{i-1}|}{\Delta t} > \Delta_{max}$.

By applying the rules above, both systematic and random sensor errors are detected and then reported by the SRV sub-service, thus reducing data uncertainty.

**SRV service interface**: The SRV service uses a clearly defined interface to interact with the application. First, a configuration command is provided so that the application can determine and adjust the service parameters ($N$, $\sigma_{max}^2$, $\sigma_{min}^2$, $LLB$, $LUB$, $TLB$, $TUB$, $\Delta_{max}$, $\alpha$, $\beta$) anytime, since they are sensor specific and are influenced by the deployment and application requirements. Second, the SRV service provides a read command which is a call to get a new sensor reading. The application receives back both the final reading ($\nu$) and a validity field. The validity field indicates whether the reading is valid (i.e., if it passes the above rules), or, if it is invalid, which type of failure(s) was detected.

Using this simple interface, the application layer on each node has full control of what is and is not sent back to the server. REDFLAG simply provides the on-node application with the data it requests along with the validation field. The application layer may filter messages based on what it sees. For example, the application may wait for $N$ erroneous readings before alerting the server (with a single message) or it may choose to alert the server on the first erroneous reading and then not send any more until the error is cleared. In either case there is a significant decrease in network overhead – the most costly aspect of the WSN. Therefore, a small amount of data processing on the node using SRV and a thin application layer can potentially have a dramatic reduction in resource cost (in terms of battery power). Also, as we described in the SRV algorithm, the SRV service takes multiple samples and then computes an average reading which theoretically increases the quality of each data point.

**SRV service parameter determination**: SRV is designed to be adaptive to dynamic WSN conditions, such as sensor calibration shifts or changing surrounding conditions. Here, a simple methodology is described by which initial service parameter values may be determined prior to WSN operation. It is recognized that even the most careful planning may still result in a set of service parameters that cause the SRV sub-service to behave undesirably (e.g., it may report too many false negatives). Consistent with our flexibility guarantee, all service parameters can be adjusted at will by the application during WSN operation using available SRV configuration commands.

- $\sigma_{min}^2$: $\sigma_{min}^2$ 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.

- $\sigma_{max}^2$: Due to noise influence, contiguous sensor readings might show some variability. $\sigma_{max}^2$ 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, may also be used to calculate an expected $\sigma_{max}$. For example, if the sensor has an accuracy of $\bar{x} \pm \delta$, one could assume that this interval, $|x - \delta, x + \delta|$, contains 95% of the expected readings for a given variable of interest. Using a Gaussian noise model, $\sigma_{max} = \delta/2$.

- $LLB$ and $LUB$: 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.

- $TLB$ and $TUB$: Often times, the range in which the calibration is valid is more strict than the total reading range. 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, $\bar{R}_{i+1}$ and $\bar{R}_i$ over a period of $t_i$ time units, one may compute: $\Delta_{E} = \max, E \left[ \frac{|\bar{R}_{i+1} - \bar{R}_i|}{t_i} \right]$. $\Delta_{max}$ should then be chosen such that $\Delta_{max} \geq \Delta_{E}$.

---

1 By “application”, we refer to the application layer on each node. Note that REDFLAG resides on each node between the application layer and the network layer as described in Section 1.
I. Urteaga et al. / Pervasive and Mobile Computing 5 (2009) 432–446

Timeline of the NSR algorithm.

• $\alpha$ and $\beta$: These calibration parameters can be adjusted for each sensor at any time. In this paper only linear calibration is considered, but REDFLAG may be easily modified to work with any calibration model. The impact of sensor calibration drift on data can be minimized by carefully adjusting $\alpha$ and $\beta$ parameters as it occurs. If calibration drift becomes significant, the reported readings will be erroneous using the original calibration parameters. In some cases, the drift is deterministic enough to apply a drift model so that the calibration parameters can be adjusted automatically without re-calibration.

For our specific application of subsurface contaminant monitoring, please refer to Section 4 for discussions about how these parameters are selected.

3.2. 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 [18] presented and validated a 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 [18] 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 [18], 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. These goals are 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.

Overview of the NSR algorithm: The algorithm operates using a tunable timing schedule depicted in Fig. 2. Clock synchronization among nodes is assumed, which can be achieved using an existing algorithm [21]. $T_d$ (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 ($T_r$) before the next detection epoch. $T_r$ (reporting period) is specified by the application to inform REDFLAG how often alarm reports are needed. All these epoch durations can be customized to meet specific detection latency and duty cycling requirements.

The distributed 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 (i.e., possibly dead). 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 $k_1^j$ and $k_2^j$, 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. The remaining energy information is piggybacked into every message sent by a node; the link quality between nodes \( n_i \) and \( n_j \) is computed by node \( n_i \) when receiving a packet from node \( n_j \).

A node is required to keep a minimum of \( NS_{\min} \) and a maximum of \( NS_{\max} \) number of nodes in its neighbor table. Because of memory limitations, a node should not monitor too many nodes. At the same time an 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 node is considered \( n_i \)'s \( j \)th neighbor if the link quality, \( A_{i,j} \), is greater than \( A_{\min} \) (the Link Quality Threshold). \( A_{\min} \), 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 1. 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. Also note that REDFLAG does not provide any message scheduling mechanism, so it relies on the underlying MAC layer protocol to resolve any potential packet collisions. 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 \( n_j^i \), if \( A_{i,j} \) is greater than \( A_{\min} \).

Once this occurs, \( n_i \) enters into the FSM for node \( n_j^i \), 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. \( k_1[j] \) counts down (from \( k_{1,\max} \) to 0) the number of epochs that \( n_i \) has not heard from \( n_j \). Whenever \( n_i \) hears from \( n_j \), \( k_1[j] \) is reset. If node \( n_i \) receives an ALR about node \( n_j \), \( k_1[j] \) is updated with the larger value between its current value, \( k_1[j] \), and the value received in the message (say, from neighbor \( n_k \), \( k_1[k] \)). Updating \( k_1[j] \), instead of resetting it, reduces detection latency and avoids potential loops. While in this state, \( n_i \) rejects all received ALQ messages about \( n_j^i \) by broadcasting an ALR response. Finally, if \( n_i \) does not hear from \( n_j^i \) in \( k_{1,\max} \) 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 \( k_{2,\max} \) 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 \( k_2[j] \). Upon obtaining an ALR from, say, node \( n_k \), node \( n_i \) transitions back to the Local Detection state and adopts \( k_2[k] \) as its own. It also acknowledges that it has received this information by transmitting an ACK to \( n_k \). If \( n_k \) receives the ACK from \( n_i \), no more ALRs will be sent in the following epochs to \( n_i \). However, if no ACK is received because either the ALR or the ACK was lost, \( n_k \) will send a new ALR message to \( n_i \) in the following epoch. Note that, in this state, \( n_i \) does not respond to ALQs about \( n_j^i \) since it is also in doubt about \( n_j^i \)'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 \( n_j^i \)'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 \( n_j^i \) by removing it from its NT.

This logic is followed by a node \( n_i \) for each neighbor in its NT (\( n_j^i \)), 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 an 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 \( T_r \); 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 \( k_1[j]/k_{1,\max} \) and \( k_2[j]/k_{2,\max} \)), and the estimated root cause of failure. Root causes detected by the

<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_k ))</td>
<td>Alarm rejection</td>
<td>Broadcast by node ( n_i ) to indicate that doubtful node ( n_k ) is still alive (remaining ( k_1 ) included)</td>
</tr>
<tr>
<td>ACK (( n_i, n_m ))</td>
<td>Alarm rejection ack.</td>
<td>Unicast by node ( n_i ) to node ( n_m ) to acknowledge the reception of ( n_i )'s ALR about doubtful node ( n_j )</td>
</tr>
</tbody>
</table>
presented subservice 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.

- **NS\textsubscript{min}, NS\textsubscript{max} and \Lambda\textsubscript{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 node running NSR. Note that application-specific \Lambda\textsubscript{min} is used as an initial estimation, but will then be dynamically adjusted on a node by node basis.

- **k\textsubscript{1}, k\textsubscript{2}, T\textsubscript{d} and T\textsubscript{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\textsubscript{d}. The detection latency for the Final Decision of the NSR algorithm is (k\textsubscript{1}\textsubscript{max} + k\textsubscript{2}\textsubscript{max})\textsubscript{T}\textsubscript{d}. Therefore, in order to detect nodes which have been unresponsive for \textsubscript{T}f time units, T\textsubscript{f} \geq (k\textsubscript{1}\textsubscript{max} + k\textsubscript{2}\textsubscript{max})\textsubscript{T}\textsubscript{d} must be fulfilled. It is recommended to avoid usage of small k\textsubscript{1}\textsubscript{max} and k\textsubscript{2}\textsubscript{max} values; otherwise, temporary failures may be reported as permanent. T\textsubscript{n} is dependent on the maximum latency of the MAC layer being utilized, τ\textsubscript{MAC\textsubscript{Max}}, and the neighborhood size: T\textsubscript{n} \geq τ\textsubscript{MAC\textsubscript{Max}} \times N\textsubscript{S\textsubscript{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\textsubscript{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\textsubscript{d} and the node can be put to sleep until then. Sleeping period T\textsubscript{s} is then computed based on T\textsubscript{d} = T\textsubscript{s} + T\textsubscript{n}.

- **NSR reporting period (T\textsubscript{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\textsubscript{r} = T\textsubscript{d}), while other applications may just be interested in periodic information matching its data collection schedule (T\textsubscript{r} \geq T\textsubscript{d}). Note that the reporting period is bounded by the detection epoch and desired detectable failure duration: T\textsubscript{d} \leq T\textsubscript{r} \leq T\textsubscript{f}. By careful selection of T\textsubscript{r}, the application layer is able to separate or mix data and alarm report traffic loads. The only restriction is that T\textsubscript{r} \geq T\textsubscript{d} \geq T\textsubscript{n}, since it takes T\textsubscript{n} to collect information about the network.

For our specific application of subsurface contaminant monitoring, discussions on how to select these parameter values in our simulation environment and in a real deployment can be found in Sections 4.2 and 5 respectively. 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, and data collection time, etc. Recall all those parameters are subject to online dynamic updates in order to adjust to changing conditions.
4. Performance evaluation

The REDFLAG fault detection service has been implemented in TinyOS 2.x [22], 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 [23] – a TinyOS simulation tool which realistically reproduces WSN physical and link layer features [24]. 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 [25], 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 [26] that are typically adopted by various sensing devices. The energy consumption metric provides a more relevant measurement of bandwidth overhead introduced by the algorithm, as energy consumption is proportional to communication overhead [27].

4.1. 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. [8] and by Porta [4]. In [8], 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; and
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 included, simulating real sensor operation.

**Unresponsive node failure creation:** WSN deployment conditions are accurately simulated by TOSSIM 2.x [24]. 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 node 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 node 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.
4.2. 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 on different platforms is: 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 Fig. 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 nodes are powered by two AA alkaline batteries (18000 J) and take the most energy consuming case (500 mJ in 8 minutes) from the set of results, the network lifetime continuously using REDFLAG is 200 days.

**Performance of the sensor reading validity subservice:** In our simulation environment, \( \sigma_{\text{min}} \) was chosen to be close to zero due to the coarse resolution of the sensors used in our proof-of-concept study in an intermediate-scale tank [4]. \( \sigma_{\text{max}} \) was set to be 5.2, selected using the 95% confidence interval issued by the sensor manufacturer. Similarly, TLB and TUB are taken from the sensor user manual and they are 0 and 5000 respectively. LLB, LUB, \( \alpha \), and \( \beta \) were selected according to the ex situ linear calibration given in [4] and they are 27, 128, 0.78, and \(-27.84\) respectively. \( \Delta_{\text{max}} \) was set to be 0.1, determined by adding the tolerance from the 95% confidence interval to the expected change in sensed values based on simulative data. The average percentage of detection for each type of fault is given by the light-colored bars in Fig. 5, using these initial parameter values. While “out of range” 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 (0.01, 2.7, and 0.02 respectively) to improve performance. 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

---

2 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 when \( \sigma_{\text{min}} = 0 \). However, false alarms would increase dramatically if other values are used for \( \sigma_{\text{max}} \) because of the coarse resolution of the sensors.
application; for instance, by using predictions from a model (e.g., [10]). Again, REDFLAG is providing a detection service to the application layer, which is in the best position to update the optimal 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.

**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 ($N_{\text{min}}$ and $N_{\text{max}}$) and the link quality threshold ($A_{\text{min}}$) 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. $T_{\text{n}}$ is related to the maximum delay of the MAC layer ($\tau_{\text{MAC}} = 10$ ms in this case), that is: $T_{\text{n}} \geq \tau_{\text{MAC}_{\text{max}}}$; $N_{\text{max}}$; $T_{\text{n}} = 100$ ms. In order to let the nodes go to sleep, $T_{d}$ was fixed to be greater than $T_{n}$; here, $T_{d} = 1000$ ms ($T_{d} = 900$ ms). 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.

$k_{1}^{\text{max}}$ and $k_{2}^{\text{max}}$ remain to be selected. From Fig. 6, it can be seen that the optimal detection accuracy is obtained when $T_{r} \geq (K_{1} + K_{2})T_{d}$ (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 $k_{2}^{\text{max}} > k_{1}^{\text{max}}$ (the left upper triangle). Lastly, from the final row, larger values of $k_{1}^{\text{max}}$ and $k_{2}^{\text{max}}$ yield fewer reports. This analysis leads to the selection of $k_{1}^{\text{max}} = 4$ and $k_{2}^{\text{max}} = 6$. As a matter of fact, the approach presented in [18] is similar to a special case of REDFLAG where $k_{1}^{\text{max}} = 1$, $k_{2}^{\text{max}} = 1$, and $T_{f} = 0$, which has here been shown to yield inferior performance.

Using these parameter values, detection and diagnosis accuracy of the presented NSR algorithm is evaluated in different scenarios. In Fig. 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.\(^3\)

Fig. 8 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 nodes at the perimeter of the faulty area, not nodes in the middle.

Fig. 9 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.

**5. REDFLAG application**

Very recently, the hydrology community has become increasingly interested in sensor networks, which involve collecting hydrological data *in situ* without manual sampling and analysis (e.g., [1,4,6,7]). Although complex phenomenon such as subsurface contaminant transport may arguably still be undersampled, data is being made available as never before and it is anticipated that practitioners will soon be burdened with doing data-rich rather than data-poor research.

Computational models of hydrologic processes are used to calculate health risks, design contaminant cleanup strategies, guide environmental regulatory policy, and determine culpable parties in lawsuits. Such use culminates in these models dramatically influencing environmental decisions involving large investments of effort and funds. Given this, it is important for these models to be used carefully in decision making processes and it is commonly believed that models will improve as more data becomes available from emerging technologies. However, the effect that erroneous data may have on simulation results is largely overlooked – especially in light of the above pioneering WSN studies. REDFLAG was recently utilized to evaluate how subsurface contaminant transport models are hindered by erroneous data ([3]). More specifically, Barnhart et al. [3] investigated the outcome of non-linear parameter estimation under different data contexts. Since fault detection was only subsidiary to that work, here we provide additional insight specifying REDFLAG’s impact. The approach in [3] is first briefly described.

Computational partial differential equation models were used to create a realistic three-dimensional evolution of contaminants in the subsurface, see Fig. 10 for a concentration snapshot 238 days into the simulation. 100 random sensor locations were selected in the domain, and the concentration profiles, or breakthrough curves (BCs), comprised the base

---

\(^3\) The computed standard error would be much smaller if more simulations were performed, but this was not undertaken due to limited computational resources.
Fig. 6. Impact of $k_1$ and $k_2$ on NSR detection accuracy (row 1), average power consumption (row 2), and number of final reports (row 3) for 25, 64, and 100 node grid networks (columns 1, 2, and 3).

synthetic data set. Random noise and faults were added to the base synthetic data set according to that described in Section 4.1. Three data sets were derived using fault creation: (i) daily WSN data containing only theoretical noise levels according to sensor specifications, (ii) data containing the same noise and additional WSN faults, and (iii) data with fewer WSN faults due to the use of REDFLAG. These are referred to as the noisy, faulty, and filtered data sets, respectively. Values from these data sets were assimilated into a simpler contaminant transport model (see Fig. 10) every 92 days as a simulation of three different scenarios: (i) model performance using data with only theoretical noise, (ii) model performance with faults as seen in field studies, and (iii) model performance when REDFLAG is used to help detect data abnormalities.

The faulty data set contains random faults, random noise, and calibration drifts (see Section 4.1). A fault persists for a random period of time after which the faulty sensor is “fixed” (presumably by a network technician) and the sensor again reports sensible values. The filtered data set was created by passing the faulty data set through REDFLAG. REDFLAG’s messages are processed at the base station in a manner which mimics the attention such messages might get if employed in a contaminant monitoring field setting: if a sensor reading is flagged as faulty, that data was not included in the filtered data set – effectively filtering out likely erroneous data. In this case, the fault may be “fixed” more quickly than in the faulty data set because a warning message was issued to a network technician. Thus, data will be missing from the filtered data set for a shorter random period of time.
For illustration, the breakthrough curves for one of the nodes is given in Fig. 11. Several types of errors are apparent in the faulty breakthrough curve: lost packets from poor network link quality, longer periods of missing data caused by node failure, abnormally noisy data, and abrupt shifts in readings. REDFLAG was able to identify most of these at the cost of filtering out additional data.

A non-linear optimization software, called PEST ([28]), was used to find a set of transport model parameters which minimizes an objective function, $\phi$, containing the residuals between the data “received” from the WSN and the data computed by the predictive computational model. A model’s predictive performance is dependent on being able to match the models outputs to the actual observations. As seen in Fig. 12, $\phi$ was reduced by almost three orders of magnitude due to the removal of a large majority of faulty data, bringing the $\phi$ values for the filtered data set quite close to the data set containing only theoretical noise levels. It is apparent that the automatic detection of data abnormalities is crucial to the
success of subsurface contaminant transport models and probably a myriad of other scenarios where non-linear models are calibrated to real world data. REDFLAG was not only effective but also simple to apply because of its layer independence and tunability.
The NSR parameter selection in a real deployment of subsurface contaminant monitoring. Our discussions in previous sections indicate that the SRV parameters are independent of network topology, so the SRV parameters values used for our simulation environment in Section 4.2 can still be used for field deployment. However, the NSR parameters are deployment-specific. Therefore, we next provide some guidelines on how to select the NSR parameter values in a field deployment of sensor networks for subsurface contaminant monitoring.

Each node in the network should have at least one neighbor to ensure network connectivity, so $NS_{\text{min}}$ can be set to be 1. Since a sparse network is typically used for this application, 4 is a good value for $NS_{\text{max}}$. An initial value for $A_{\text{min}}$ (Link Quality Threshold) will need to be identified based on the specific environment the network is deployed. For epoch durations, the following guidelines can be used. (1) The speed of contaminant movement underground is typically very low due to underground soil heterogeneity. For instance, for a 16 feet long tank in our experiments, it takes the contaminant plumes two weeks to travel from one end to the other end of the tank [4]. Therefore, $T_d$ can be set as two hours or even larger. (2) $T_n$ is related to the maximum delay of the MAC layer ($\tau_{\text{MAC}}$) and the maximum number of neighbors ($NS_{\text{max}}$). $\tau_{\text{MAC}}$ can be set to be 10ms depending on the MAC protocol used for chosen nodes. Therefore, $T_n = 40 ms$. (3) $T_r = T_d - T_n$, which means that nodes will be sleeping most of the time, a desirable feature for long term monitoring applications in order to conserve energy. (4) The reporting period, $T_r$, can be set to be the same as the detection period (i.e., two hours), so that failures are reported as soon as they are detected. Other NSR parameters such as $k^1_{\text{max}}$ and $k^2_{\text{max}}$ can be determined using the similar approach as discussed in Section 4.2.

6. Conclusion

REDFLAG, a Run-time, 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. The methodologies we have provided for the selection of the parameters for the two subservices have been successfully applied in our empirical studies, which should be applicable and very useful to any users of REDFLAG. In order to be a generic service, REDFLAG simply allows the application access to these parameters, but REDFLAG itself has no knowledge of how to tweak its own parameters. We, therefore, leave automatic parameter selection to the application, in order to match its specific environment and requirements.

Once optimal service parameter determination is achieved, failure detection accuracy of both SRV and NSR is shown to be highly satisfactory. Application of REDFLAG to study the impact of data uncertainty on the performance of subsurface contaminant transport models further validates the benefits of using REDFLAG. Simulation results also indicate that 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 [29]) 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 new specific applications, such as in [3]. We are in the process of constructing a sensor-instrumented aquifer tank filled with sand to mimic the real world underground environment. Once the tank is ready, we will integrate REDFLAG with the wireless sensor data acquisition software and subsurface contaminant transport model running at the server. We will be able to evaluate the usability and the performance of REDFLAG in a closer-to-real environment. We hope that this approach to fault detection will prove useful in many other real WSN applications.

References


