Feluda: Provenance-enabled Diagnosis of Elusive Network Failures in Wireless Sensor Networks

S. M. Iftekharul Alam, Sonia Fahmy
Purdue University, E-mail: {alams, fahmy}@purdue.edu

***Abstract*—Sensor nodes are prone to failures due to their limited hardware capabilities, and software uncertainties stemming from erroneous logic or configuration. Such failures as well as wireless channel dynamics can degrade network performance, potentially creating network partitions. Existing troubleshooting tools either only diagnose a few problems or suffer from high overhead due to periodic transmission of control packets. In this paper, we propose Feluda[1], a system that exploits *provenance*, *i.e.,* forwarding path of data packets, for automatic localization of problematic nodes and packets. Unlike existing methods, Feluda extracts necessary network performance metrics from *packet headers* and stores them into node flash storage, thereby reducing out-of-band packet transmissions. Once problematic nodes and corresponding packets are identified at the base station (BS), Feluda provides efficient querying mechanisms to retrieve packet headers of interest from specific nodes. Packet header analysis reveals the root cause of the problem. We implement Feluda using Java and ContikiOS on the BS and sensor nodes, respectively. Testbed experiments and COOJA simulations demonstrate the effectiveness of Feluda compared to the state-of-the-art.**

I. INTRODUCTION

With limited sensor hardware capabilities and software uncertainties stemming from erroneous logic or configuration, sensors are prone to failures such as crashes, reboots, or unresponsive radios [1]. An automated troubleshooter helps network administrators understand and diagnose problems. Troubleshooting wireless sensor networks (WSNs) strives to: (i) enhance visibility into the network with fine-grained diagnosis, and (ii) minimize associated energy consumption. Today's diagnosis applications, however, cannot effectively balance these conflicting objectives. Many troubleshooting applications [2], [3] periodically collect system metrics (*e.g.,* battery power, number of neighbors, number of dropped packets) from sensor nodes, and provide fine-grained diagnosis at the base station (BS). These approaches, however, are energy-inefficient for large-scale sensor networks due to the overhead associated with pro-active information collection. Their diagnosis accuracy may deteriorate as the transmission of metrics itself can fail over the unreliable network. In contrast, methods based on local diagnosis [4], [5] have every sensor monitor its local state and neighboring nodes. While these incur lower energy overhead, they are unable to provide network administrators with the desired visibility into the network and cannot be easily extended to diagnose a wide range of network failures.

The question we seek to address in this work is: *can we achieve fine-grained but energy-efficient diagnosis of network failures*? To address this question, we design Feluda, a tool that exploits *packet headers* and *provenance* [6], [7] to troubleshoot "elusive" network failures. The design of Feluda is inspired by a recent finding for IP networks: a single packet's history can provide evidence to diagnose network failures [8]. In a multi-hop sensor network, *provenance* of a packet includes knowledge of the originator and forwarding path of the packet. The *header* of a packet, observed at different forwarding nodes, contains information (*e.g.,* parent node, link quality, routing cost, flags indicating imminent congestion) that can explain the reason behind network failures. The Feluda BS tracks packets that indicate symptoms (*e.g.,* lost/delayed/duplicate packets) and analyzes provenance embedded into the packets to localize the failure. Feluda running on the sensors efficiently stores sent/received packet headers into flash memory and sends headers of interest to the BS on demand. By requiring only a few sensor nodes (operating in the vicinity of network failures) to transmit a small number of packet headers, Feluda gives packet-level visibility at a low energy cost.

Feluda is a stand-alone module, independent of the data collection application running on sensor nodes. Feluda exploits a standard sniffer interface to collect packet header information, unlike existing solutions [2], [3] which collect information through instrumentation of the application and networking stack. It is known that writing a byte to flash is 20$\times$ more energy-efficient than transmitting [9]. Feluda consolidates headers of retransmitted data and corresponding ACK packets into a single entry that is written to flash, thereby reducing the number of flash memory accesses. To provide faster insertions and querying over a large number of entries, we use energy-efficient MaxHeap indexing [9] built on a file system [10] with a small memory footprint. Finally, we design a protocol to send/respond to queries for specific packets to the neighboring nodes of a suspect node. This avoids using the sensor data collection protocol because the protocol itself can be plagued with configuration errors or other failures.

We implement the Feluda sensor node module using ContikiOS [11] and the BS application using Java. We consider failures such as node down, soft reset, and congestion. We observe that a node with incorrect configuration can cause routing changes, packet loss, or network partitions. For example, a node mistakenly configured as a BS may partition a network. Through testbed experiments and simulations, we demonstrate that Feluda effectively diagnoses these failures with lower energy consumption than the state-of-the-art.

To summarize, this paper makes the following contributions: **(1)** We design Feluda, the first (to the best of our knowledge) sensor network troubleshooting system that demonstrates how *provenance* and *packet headers* can be used to diagnose a wide range of network failures with reduced

[1]Feluda is a popular fictional detective in Bengali literature who is known for his analytical abilities and observation skills.

energy consumption. **(2)** We present a BS-side algorithm which monitors data packets over a sliding window and correlates their provenance to detect and localize network failures. **(3)** We devise a query propagation protocol (independent of sensor data collection) to transmit queries for specific packets from the BS to a particular neighborhood. This ensures uninterrupted transmission of query and response messages in the presence of network failures. **(4)** We identify a wide range of configuration errors that can lead to mild (*e.g.,* route changes) to severe (*e.g.,* network partition) network failures. **(5)** We implement Feluda for sensor nodes to use a MaxHeap [9] index to ensure energy-efficient querying and storage of packet header information. **(6)** We evaluate Feluda via COOJA [12] simulations and a real-world testbed of 25 TelosB motes. The results demonstrate that Feluda effectively diagnoses network failures with 24-57% lower energy consumption compared to the state-of-the-art.

II. Related Work

Sympathy [2] is one of the earliest tools to debug failures in wireless sensor networks. Sympathy requires each node to actively transmit control packets (with network connectivity and flow information) towards the BS. The Sympathy BS application checks if insufficient data has been received from a node and determines the root cause of packet loss using an empirically constructed decision tree. VN2 [3] is a recent diagnosis tool, which periodically collects several metrics from sensor nodes. The VN2 BS then attributes symptoms to one or more root causes. By analyzing historical information with a Non-negative Matrix Factorization model, it trains a matrix of network exceptions where each row represents potential root causes of network exceptions. AD [13] also collects system metrics and discovers silent failures by analyzing correlation graphs of the metrics. These approaches, however, are energy-inefficient for large-scale sensor networks due to the overhead associated with their pro-active information collection strategy.

To reduce the overhead of active information collection, a number of approaches [14], [15] use in-network packet tagging to deduce the forwarding path of packets and detect path changes in the network. By building a probabilistic inference model representing dependencies among nodes [14] or querying nodes involved in path changes [15], they diagnose a limited set of problems, namely node or link failures. While Feluda takes a similar approach (using provenance to localize failures), it exploits packet headers to diagnose a wider range of network failures and offers enhanced visibility into the network. LiveNet [16] and Dustminer [17] are two methods that also avoid pro-active information collection and offer good visibility. LiveNet, however, uses external sniffer devices to snoop packets forwarded by sensor nodes and analyzes them at the BS to understand network dynamics. Dustminer records network and kernel-level events into local flash by instrumenting corresponding handler routines and uses a frequent pattern mining algorithm over these records to diagnose failures. In both approaches, all information logged into flash is transferred to the BS as a batch. In contrast, Feluda only requires a few sensor nodes to transmit a small amount of logged information to the BS, only in cases of failures.

In contrast to BS-based approaches, methods based on local diagnosis [4], [5] have every sensor monitor its local state and neighbors. Neighbors cooperate to diagnose exceptions. While these methods incur low energy overhead, they cannot provide administrators visibility into the network and also cannot be easily extended to diagnose a wide range of failures. Source level debuggers [18], [19], [20] debug programming errors in network protocols and are orthogonal to our approach. Netsight [8] is a recent IP network troubleshooting platform that inspired us to exploit packet histories to diagnose network failures. Like many existing IP network debugging tools, direct realization of Netsight is impossible for resource-constrained sensor networks.

III. System Model and Challenges

In this section, we discuss our system model and types of network failures, and illustrate the role of provenance and packet headers in localizing failures.

A. Network Architecture

We consider the archetype WSN applications where all nodes forward data to a BS in a multi-hop fashion. Within a particular time window, independent observations obtained at the BS from different sensors are concerned with the same event. The data forwarding process is enabled by best-effort data collection protocols such as CTP [21] or Contiki Collect [22]. Data is typically forwarded along a minimum cost tree rooted at the BS. The routing cost is expressed in terms of the number of expected transmissions (ETX). Routing beacons are broadcast periodically in a neighborhood to keep the ETX estimates up-to-date.

B. Network Failures

We identify three types of node faults that affect network performance: (1) the resource-constrained hardware, (2) software bugs, and (3) configuration errors. When the battery of a node becomes low or one of its hardware modules fails, the node becomes unresponsive and disconnected from the network. Incorrect initialization of the radio software module [1] or faults in the radio stack [18] may cause the same problem. Another common software bug is when some tasks are starved due to a long-running task, resulting in overflow of the task queue and continuous software reset [23]. When a node goes down or resets, data packets from the node itself and its descendants may experience loss, delay or route changes, thereby disrupting network operation.

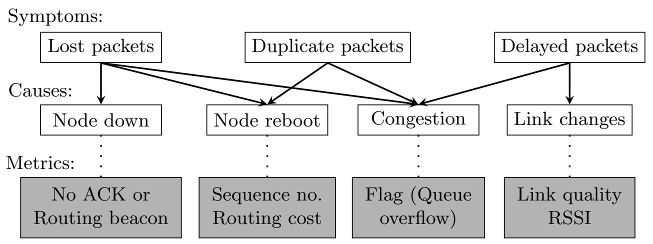
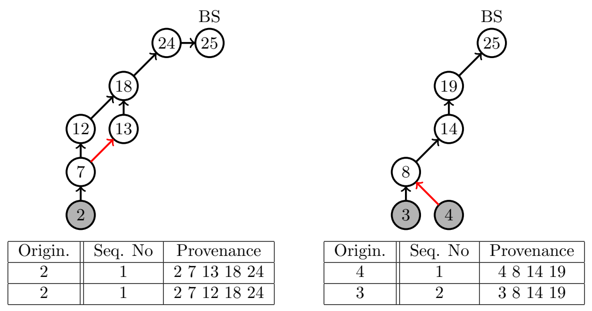


Fig. 1: Metrics for root cause identification

Data collection applications have configuration parameters such as "sink"/"root", maximum data length, and transmission power. Data dissemination protocols are used to send configuration updates from the BS to sensor nodes [24]. Incorrect parameters may cause network problems. When a node is incorrectly configured as a root node, it no longer forwards data from its descendants to the actual BS and thus creates a network partition. Changing transmission power may lead to topological changes in the network. An incorrect maximum data length may prevent a node from transmitting data leading to packet loss. Apart from these errors, environmental changes and unreliable wireless channels may cause the MAC layer to drop packets after the maximum retries. The top level nodes with many descendants are also likely to experience congestion and may drop packets due to lack of buffer space.

A BS only observes a handful of abnormal events or *symptoms* of a failure, such as high inter-packet delay, packet loss, or reception of duplicate packets. Fig. 1 relates possible root causes of network failures with their corresponding symptoms. For example, with throughput degradation, symptoms include lost packets, duplicate packets and high inter-packet delay. These symptoms are caused by a number of network failures, *e.g.,* packets can be lost due to node failure, link failure or congestion in the network. Our goal is to find root causes of frequently observed symptoms.



Origin.	Seq. No	Provenance
2	1	2 7 13 18 24
2	1	2 7 12 18 24

(a) Duplicate packets

Origin.	Seq. No	Provenance
4	1	4 8 14 19
3	2	3 8 14 19

(b) Lost packet

Fig. 2: Application of provenance to localize suspect nodes

C. Packet Headers and Provenance

In typical data collection applications, three types of packets are exchanged among sensor nodes: data, ACKs, and routing beacons (ADVs). The packet headers contain useful information or *metrics* about network interaction among a set of neighboring nodes. For example, the header information in a data packet received at a node includes (but is not limited to) sequence number, sender, routing cost, and flags indicating potential congestion. Fig. 1 shows some packet header information (metrics) that can be useful to diagnose problems. The headers of a packet observed at different nodes along the forwarding path represent the complete history of the packet throughout the network. Although the collection and analysis of packet histories was proposed to troubleshoot traditional wired networks [8], direct realization in resource-constrained sensor networks is challenging. If packet headers are transmitted periodically, the transmission energy would be prohibitive. Fortunately, provenance helps us tackle this problem.

In a multi-hop network, provenance of a packet includes knowledge of the originator and path. We can leverage energy-efficient methods to embed and transmit provenance into data packets [7]. The BS uses provenance to locate nodes potentially responsible for network failures. Once a node is identified, we can collect and analyze packet headers from its neighborhood to diagnose failures. This tackles the issue of energy consumption by transmitting only a few packet headers on-demand, while gaining visibility into the network at the granularity of a packet.

D. Motivating Examples

To better understand how provenance and packet headers can be used to localize and diagnose failures, we conduct a series of testbed experiments with 25 TelosB motes. We consider a 5×5 grid network where each node sends a packet every 2.5 seconds. Every node stores the header of each received or transmitted data/ACK/routing packet into local flash. Each data packet is uniquely identifiable using a sequence number provided by the data collection protocol. Every forwarder node embeds its ID into data packets as provenance. We use a Java-based BS application to receive data packets through the serial port, collect provenance information, and detect symptoms such as duplicate, delayed, or lost packets.

Fig. 2 illustrates two cases for duplicate and lost packets at the BS. Fig. 2(a) shows that two packets have been received from node 2 with the same sequence number but different provenance: {2,7,13,18,24} and {2,7,12,18,24}. This indicates potential link changes from node 7. We investigate the sequence of packet headers stored at node 7 and find that 7 did not receive a number of ACKs from 13 which in turn, changed the link quality estimate from node 7 to 13. Eventually, node 7 chose another node 12 as its best next hop. Fig. 2(b) depicts a case where the packet with sequence number 2 from node 4 is lost. However, a packet with the same sequence number from another source node 3 (representing the same event as the packet number 2 from node 4) is received with provenance: {3,8,14,19}. If we correlate this provenance information with that ({4,8,14,19}) of the last successfully received packet from 4, we suspect the link between 4 and 8 is a problematic one. The corresponding packet headers stored at node 4 reveal that packet number 2 was dropped after the maximum retries. These results motivate us to exploit provenance to localize suspect nodes/links, and store and analyze packet headers to diagnose network failures with fine-grained visibility.

E. Challenges

Managing provenance, storing packet headers into flash and querying them face non-trivial challenges:
1. **Provenance analysis**: We need to correlate provenance across different events at the BS to localize suspect nodes. This requires populating and matching a number of graphs each consisting of all nodes present in the network. The process must scale as the network size increases.
2. **Flash memory management**: Flash memory has a constraint that a part of the memory must be erased before overwriting contents [10]. Erasing should be performed only when necessary; otherwise, excessive erasing may cause the flash to wear out. An efficient indexing mechanism is also required to quickly scan through stored records. A general-purpose sensor database system [9] is thus implemented over a special file system. It is, however, an overkill to use this generic system solely for storing packet headers as it uses significant code (ROM) and main memory (RAM).
3. **Query-response mechanism**: Once a suspect node is identified at the BS, we need to send a query message to the

neighborhood of the suspect asking for header information of specific packets. Since the data collection protocol maintains routing paths *towards* the BS, a separate best-effort mechanism is required to propagate the query from the BS to the designated nodes. For sending query responses to the BS, a node cannot rely on the data collection protocol since its parent node can be faulty due to configuration errors.
4. **Energy cost**: Though writing a byte to flash is 20× more energy-efficient than transmitting [9], we should still keep the number of read/written bytes (or, accesses) as low as possible.

IV. FELUDA DESIGN

Feluda adopts two design philosophies: (i) localizing a failure is key to finding the true cause of failure [2], and (ii) packet history provides evidence to diagnose network problems [8]. Feluda conducts scalable provenance analysis at the BS to localize problems. By enabling sensor nodes to efficiently store packet headers and requiring only a few to transmit a small number of packet headers, Feluda maintains packet-level visibility at a low energy cost. Fig. 3 shows the components of Feluda for the BS application and sensor nodes. We provide details of these components in the subsequent sections.

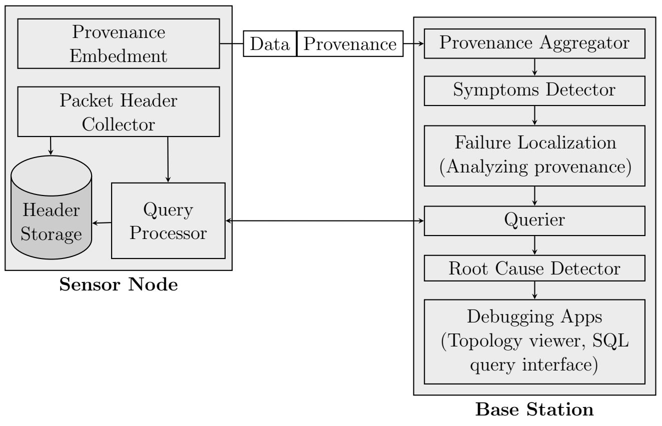


Fig. 3: Architecture of Feluda

A. Base Station Application Components

Each data packet has a unique sequence number set by its originator. Since sensor networking applications have very low data rates, a limited number of bits (*e.g.,* 16) is sufficient for sequence numbers. Packets originating from different nodes but having the same sequence number are assumed to represent the same physical event. Packet $p_s^{(n)}$ from node n with sequence number s is considered to represent event e_s. Since data from different nodes can take different paths and experience variable delays to reach the BS, the application buffers packets corresponding to the last few (*e.g.,* 10) events before processing them.

1) Provenance Aggregator: We extract provenance data from a packet using a decoding method [7] compatible with the encoding method used to embed provenance during forwarding. We construct provenance of a packet, $p_s^{(n)}$ as a graph, $G_s^{(n)}$. Provenance of data packets originating from different nodes but pertaining to the same event is merged into a larger graph, $G_s = \cup_m G_s^{(m)}$. Fig. 4(a) shows a merged graph for a 3×3 grid network where the gray colored nodes 3, 7, and 9 send packets to the BS. In this particular event, e_{50}, packets $p_{50}^{(3)}$, $p_{50}^{(7)}$, and $p_{50}^{(9)}$ have provenance $3 \rightarrow 2 \rightarrow 1$, $7 \rightarrow 5 \rightarrow 1$, and $9 \rightarrow 5 \rightarrow 1$. We will use this network as a running example when describing Feluda modules.

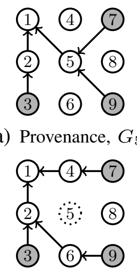
2) Symptom Detector: The *symptom detector* maintains a sliding window of events. It saves the packets received in the window and keeps 4 bitmaps per originator to track symptoms, *i.e.,* packets that are lost, delayed, duplicated, or have experienced link changes on their way to the BS. Unless otherwise stated, the window is advanced by one upon processing each event. Assume that the event being processed is e_s and the size of the window is w. The bitmap for an originator n maintains records for packets with sequence numbers $s-w+1$ to s: $p_{s-w+1}^{(n)}, p_{s-w}^{(n)} \cdots p_s^{(n)}$. If a packet $p_i^{(n)}, i \leq s$, satisfies any of the following conditions, the i-th bit is set to true:

$$\begin{aligned} &\textit{delayed}: \textit{timestamp}(p_i^{(n)}) - \min_m \textit{timestamp}(p_i^{(m)}) > \delta_{th} \\ &\textit{duplicate}: p_i^{(n)} \text{ was received before} \\ &\textit{link changes}: G_i^{(n)} \neq G^{(n)} \\ &\textit{lost}: p_i^{(n)} \text{ is not received} \end{aligned}$$

Here, a packet, $p_i^{(n)}$ is considered delayed if it arrives δ_{th} units after the arrival of the first packet denoting event i. $G^{(n)}$ indicates the provenance of the last packet that is received from node n in the previous window of events ($[e_{s-2w+1}, e_{s-w}]$) without any symptoms.

If the cardinality (number of bits set) of a bitmap reaches a given threshold, an alarm is raised to notify the *failure localization* module of a potential problem. The sliding window is then advanced by w. A large window size lengthens failure detection time, and small threshold values may prematurely flag problems. Network administrators choose the sliding window size and thresholds for different symptoms depending on the network conditions and debugging requirements. Fig. 4(c) shows the *lost* and *link changes* bitmaps for three originator nodes in our example network. The window has a size of 10 and comprises events e_{51} to e_{60}. Node 5 goes down after event e_{55} and the packets originating from nodes 7 and 9 (using 5 as the next hop) start experiencing symptoms. The *lost* bitmap for originators 7 and 9 shows that packets 56 and 57 are lost. Eventually, nodes 7 and 9 find new parents. This change is reflected in Fig. 4(b), which shows a provenance graph constructed during event e_{58}. The *link changes* bitmaps for originators 7 and 9 also record that packets 58 to 60 experienced link changes with respect to the last window. If the threshold for the combined *lost packets* and *link changes* symptoms is set to 0.5, the *fault localization* module will receive an alarm for the window $[e_{51}, e_{60}]$. This indicates that a failure is detected when at least 50% of packets in a window experience symptoms.

3) Fault Localization: The *fault localization* module handles the following cases: (1) *Duplicate packet*: If the provenances of duplicate packets are the same, the last node before the BS becomes suspect. Otherwise, the provenances of duplicate packets are compared to identify nodes that are absent in the provenance of at least one packet; (2) *Delayed packet*: If a packet $p_i^{(n)}$ is delayed, all the nodes that are present in the provenance of $p_i^{(n)}$ but absent in the provenance



(a) Provenance, G_{50}

(b) Provenance, G_{58}

Window: Start position = 51, End position = 60

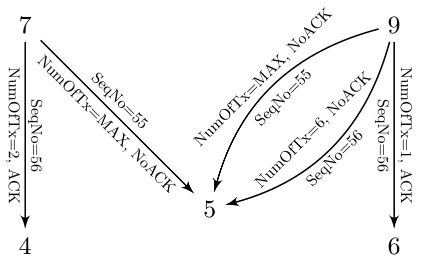
Sender	Bitmap for link changes									
3	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	1
9	0	0	0	0	0	0	0	1	1	1

Sender	Bitmap for lost packets									
3	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0
9	0	0	0	0	0	1	1	0	0	0

(c) Symptom detection

$$\begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 9 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 \end{array}$$

(d) Suspect node localization



(e) Root cause detection

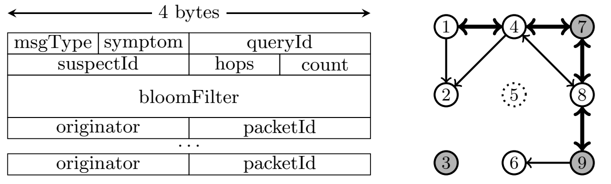
Fig. 4: Feluda operating in an example 3 × 3 grid wireless sensor network (gray nodes are data generators)

of all other packets pertaining to the event e_i are considered suspects; and (3) *Lost packets* or packets with *link changes*: These two cases are combined since a lost packet can be treated as a packet with missing links. For either of these symptoms observed in an event e_i, we construct an adjacency matrix of the combined provenance graph for this event, $Adj(G_i)$, which is a 0-1 matrix. Graph $G^{(n)}$ records the provenance of the last successfully received packet in the previous window from node n. We merge these provenances from all originator nodes into a graph $G = \cup_m G^{(m)}$ and construct the corresponding $Adj(G)$. The non-zero entries of the matrix $\neg Adj(G_i) \wedge Adj(G)$ reveal all the links that disappeared from the previously seen stable routing tree G. The nodes adjacent to these links are considered suspects if they are absent in G_i.

At the end of the window, several nodes are marked as suspect. The ID of the highest marked suspect node and the first packet denoting a symptom (called the *testimonial packet*) are dispatched to the *querier* module for root cause detection. Fig. 4(d) shows the outcome of $\neg Adj(G_{58}) \wedge Adj(G)$ for our example, indicating that links $7 \rightarrow 5$, $9 \rightarrow 5$, and $5 \rightarrow 1$ disappeared. Out of the nodes adjacent to these links, node 5 is missing from G_{58} and hence is considered suspect. Node 5 is the highest marked suspect node at the end of the window, and packets number 56 from both nodes 3 and 7 are considered testimonials.

4) Querier: The *querier* module queries the neighbors of the suspect node for the header information of testimonial packets. The query, however, cannot be sent using the data collection protocol, as it does not maintain unicast routing paths to all nodes. The data dissemination protocol [24] provides an alternative but incurs high transmission overhead since it floods packets throughout the network. Fortunately, the provenance graph of the latest event gives us currently active paths (*i.e., query paths*) from the neighbors of the suspect to the BS. By exploiting this information, we design a custom protocol (discussed in Sec. IV-B3) to propagate query messages from the BS to the desired set of nodes. The protocol requires the *querier* to embed at least one of the query paths and its hop count into the query packet. We use Bloom filters to encode the node IDs on a *query path* in a bit-efficient manner. The protocol is also used to send replies.

Fig. 5(a) shows the format of a query packet. In addition to the Bloom filter, the packet includes the type of message (query or response), the observed symptom type (*e.g.,* link changes), a query ID, the suspect node ID, the number of testimonial packets and their originator and sequence number. In our example scenario, the query includes 5 as a suspect node



(a) Query message format

(b) Query propagation

Fig. 5: Query message format and propagation in Feluda

and encodes 1, 4, and 7 (present on the *query path* $7 \rightarrow 4 \rightarrow 1$) into the Bloom filter. The packets with sequence number 56 originating from nodes 9 and 7 are testimonial packets.

5) Root Cause Detector: The *root cause detector* module analyzes responses received from the neighbors of the suspect node. The response message corresponding to a query contains the query ID and header information of testimonial packets. While headers of testimonial packets provide ample information to identify the root cause of the problem, we allow sensor nodes to include a few additional headers (at most three in our current implementation) in the responses. These are packets that have the same originator as a testimonial packet but have sequence number one less or more than that of the testimonial packet. This can give a holistic view of network dynamics before, during, and after the failure. We model the header responses specific to a query using a multi-graph which allows multiple edges between two vertices. The *source* and *destination* fields (see Table I for packet header description) of all the received packet headers form the set of vertices. Each packet header also generates an edge labeled with that packet header between respective source and destination vertices.

The multi-graph representation of query response allows network administrators to analyze packet headers from different perspectives. For example, by traversing the graph according to a particular packet sequence number, the interaction among the neighboring nodes can be observed at the time of reception of that packet. Iterating over the edges incident to a suspect node, the sequence of events leading to a network failure becomes evident. Referring back to our example, Fig. 4(e) shows packet headers received via response messages from nodes 7 and 9 and the corresponding multi-graph representation. The edges labeled with sequence number 55 and incident to suspect node 5 reveal that even after maximum retries, both nodes 9 and 7 did not receive ACKs from node 5 for packet number 55. As a result, packet 55 was dropped at both nodes 9 and 7 which also find their routing cost increased. While transmitting the packet with sequence number 56, node 7 switches to node 4. Node 9, however, still tries to

TABLE I: Description of a *log entry* which collates information from the packet headers of different network protocols

Attribute	**Attribute source**	**Description**	**Role in diagnosis process**
Originator	Data packet	The node ID that originally generates this packet	• The ⟨originator, sequence number, source, destination⟩ tuple uniquely identifies a data packet and also a *log entry* • The ⟨local sequence number, source, destination⟩ tuple uniquely identifies an ACK packet • By mapping two tuples, ACK header information can be combined with the corresponding data packet header into a single *log entry* • The sequence number also helps determine if an originator has recently rebooted
Sequence number	Data packet	This is a data collection protocol sequence number set by the originator	
Source	Data/ACK/ADV packet	The node ID that sends this packet locally	
Destination	Data/ACK packet	The node ID to which this packet is forwarded locally	
Local sequence number	Data/ACK packet	This is a MAC protocol sequence number set by the local source	
ETX	Data/ACK packet	The routing cost estimate of the source towards the BS via destination node	Aids in determining the reason of routing path (*i.e.,* link) changes
Parent's ETX	ACK/ADV packet	Parent's routing cost estimate towards the BS	Useful to check if the parent is a root, or has lost its path or rebooted recently
Hop count	Data packet	The number of hops traversed by this packet	Useful to determine if this packet has experienced a routing loop
Flag	ACK	Indicates if the corresponding data packet was received by the sender of this ACK and gives the reason if the data packet was dropped	Aids in determining the reason of lost packets and congestion
LQI	Data/ACK packet	Link quality indicator for a received packet	Speaks for any environmental changes (*e.g.,* increased noise) in the network
RSSI	Data/ACK packet	Signal strength indicator for a received packet	Useful to detect an erroneous transmission power setting of the sender
Num of TX	Data packet	Indicates MAC layer retransmissions	This along with flags explains any significant changes in routing cost
Packet length	Data packet	Indicates the length of data packet	Identifies transmission failure due to packet size exceeding maximum value

transmit packet 56 through node 5 and fails to get any ACK from node 5. After a number of retries, node 9 finds a better path through node 6 and forwards packet 56. To summarize, all the neighbors detected node 5 as unresponsive, which confirms that node 5 is down. We also develop a debugging application for the BS to allow the network administrator to issue a query to a specific node about a specific packet and visualize the responses via multi-graph. Please refer to [25] for more details.

B. Sensor Node Components

1) Provenance Embedding: If every node embeds its ID into the provenance field of the data packet being forwarded, the length of the provenance increases as the packet traverses the network. Using a fixed bit budget, we use an energy-efficient *Rabin fingerprint* provenance encoding scheme [7]. As a node receives a packet, it concatenates its own ID to the existing value in the provenance field and updates the fingerprint. The corresponding decoding scheme is used at the BS to extract provenance.

2) Packet Header Collection: There are three types of packets exchanged in the network: data packets, ACK packets, and routing advertisements (ADVs). Packet headers contain valuable information about the status of the sender. Table I gives a description of the information collected from these packet headers. This information constitutes a *log entry*. An entry is 16 bytes, and can be uniquely identified by the tuple $\langle originator, source, destination, sequence\, number\rangle$. Every node maintains a small in-memory list of log entries.

We consolidate header information of an incoming (outgoing) ACK packet into the log entry of the corresponding outgoing (incoming) data packet. If a packet is retransmitted, relevant fields (*e.g.,* number of transmissions, ETX) of the log entry specific to that packet are also updated. In case of a routing ADV, we collect headers from *incoming* ADVs, which provide the ETX of the senders towards the BS. The most recent in-memory log entries that have destination equal to the sender of an incoming ADV packet save the advertised ETX.

The in-memory list of log entries consolidates header information of different types of packets into a single entry, thereby reducing space requirements and number of accesses of flash storage. The oldest log entry in the in-memory list is stored into the flash memory until either a configurable amount of time expires, or the list becomes full. Note that storing and retrieving entries into and from flash memory should be energy-efficient and quick. At the same time, care should be taken so that erasures and re-writes are distributed evenly across the entire flash memory. While we can use a sensor node database system such as Antelope [9] to manage log entries, we find that its features are unnecessary for our purposes and they take extra space in both RAM and ROM. Hence, we only borrow Antelope's indexing mechanism for energy-efficient storage and querying.

We use a *MaxHeap* [9] index due to its wear-leveling performance and reduced energy usage. The MaxHeap uses a binary maximum heap data structure, where a heap node maintains a reference to a bucket of log entries. We index the *sequence number* field of the log entry. The memory requirement of the MaxHeap index is $O(n+4k)$ bytes, where n is the number of nodes in the heap and k is the number of keys in each node.

3) Query Processor: We design a *query propagation protocol* (see Algorithm 1) to forward a query message from the BS to the neighborhood of a suspect node. The format of the query packet is shown in Fig. 5(a). The *query propagation protocol* relies on the following four fields of a query packet: query ID, the suspect node ID, the Bloom filter marked with node IDs present on the path from a neighboring node to the BS, and hop count. While other fields are self-explanatory, hop count needs more explanation. As the query packet is propagated, the hop count is decremented. When the hop count becomes 0, packet propagation stops. In our design, the value of hop count is set to number of node IDs encoded in the Bloom filter plus one. Since the Bloom filter contains path information to reach only one neighbor of the suspect node, the addition of one allows propagation of the packet two hops away from that neighbor. This increases the chance of the packet being received by other neighbors of the suspect node [25].

Fig. 5(b) shows the propagation of a query packet in our example network. The packet indicates that the suspect node is 5, the Bloom filter encodes node IDs 7, 4, and 1, and the hop count 4. The BS broadcasts the query packet first. Nodes 2 and 4 receive the packet and node 4 rebroadcasts, while node 2 does not since it is not present in the Bloom filter. Node 4 also marks 1 as its upstream node. When node 1 receives the broadcast from node 4, it treats this an ACK from node 4. Eventually, node 7 receives the packet from node 4, and rebroadcasts after marking node 4 as its upstream node. When node 8 receives the packet from 4, it does not rebroadcast since the hop count in this case is 3 and node 8 is not present in the Bloom filter. Node 8, however, rebroadcasts the packet when it receives it from 7, since the hop count is 2 this time. In this way, node 9 receives the packet from 8 and rebroadcasts since the hop count is still 1. Nodes 8 and 9 also indicate 7 and 8 as their upstream nodes, respectively. The query propagation stops at node 6 since the hop count becomes 0. The propagation path is shown using thick edges in the figure. When nodes 7 and 9 find desired log entries, they each prepare a response message and forward it to the BS through the upstream nodes.

Algorithm 1 Query propagation protocol running at node n

```
Input: The query packet, $q$ received at node $n$
Initialization: $lastQ \leftarrow \phi$, $wasAcked \leftarrow F$, $upstreamNode \leftarrow 0$
 1: if $(upstreamNode \neq q.sender) \wedge (q.id = lastQ.id) \wedge (q.hops = lastQ.hops - 1)$ then
 2:     $wasAcked \leftarrow T$
 3: end if
 4: if $(q.id < lastQ.id) \vee (q.hops = 0)$ then
 5:     return
 6: end if
 7: $q.hops \leftarrow q.hops - 1$
 8: if $q.hops > 1 \wedge \neg q.BloomFilter.contains(n)$ then
 9:     return
10: end if
11: $lastQ \leftarrow q$
12: $upstreamNode \leftarrow q.sender$
13: if $q.hops > 0$ then
14:     Start a timer to rebroadcast $lastQ$ until $wasAcked$ becomes $true$
15: end if
16: if $q.hops \leq 2 \wedge q.suspect \neq n$ then
17:     Search and retrieve $log\ entries$ to make $response$
18:     $reliableUnicastTo(upstreamNode, response)$
19: end if
```

V. Implementation and Evaluation

We implement the Feluda sensor node module and BS application using C and Java, respectively, and conduct simulations and testbed experiments to evaluate performance. We port our implementation to the ContikiOS. In order to store *log entries* into flash memory, we adapt the MaxHeap implementation provided by the database system Antelope [9]. Our implementation of storage and query modules uses 4.7 KB of ROM, whereas Antelope uses 17.3 KB [9]. The breakdown of code size for different Feluda modules is given in [25]. We use the Contiki Collect protocol [22] to send data from sensor nodes to the BS, and expose an *intercept* interface [21] to embed provenance into data packets. We implement our query propagation protocol using two primitives (*broadcast* and *reliable unicast*) provided by the Contiki RIME [22] stack.

A. Simulations

We conduct simulations using COOJA and emulate Tmote sky motes. The sink mote interacts with the BS application via a serial interface. We compare Feluda to the most recent related work, VN2 [3], which uses periodic control packets to transmit node and link metrics (forwarding path, routing table, link quality) from every node to the BS. To make the comparison fair, we use the same provenance embedding scheme used by Feluda to encode the forwarding path into the VN2 control packets.

In our simulations, 30% of the nodes generate data items every 10 s. The data collected at the BS during a period pertains to the same event. We use 10 as the sliding window size for Feluda and 0.5 as the threshold to detect a failure. We vary the periodicity of control packets used by VN2 from 60 s to 180 s. To introduce network failures, we consider a number of faults which are injected in the 10th minute of each experiment. Each simulation is 20 minutes long. The results are averaged over 10 runs. We consider the following performance metrics: *Average Energy Consumption*: The average amount of energy consumed per node due to transmission, reception, and reading/writing flash memory (if any); and *Failure Diagnosis Time*: The time taken to diagnose a network failure. To measure total energy consumption, we use the values of voltage and current consumption specific to Tmote sky [9], [26]. We collect per node energy consumption data every 30 s.

1) Grid Network Topology: We vary the grid dimensions from 3×3 to 9×9, keeping the nodes spaced 20 meters apart and the BS in the upper left corner of the grid. We set the period of packet transmissions to 180 s for VN2. We start with an experiment in which an arbitrary forwarder node goes down. Fig. 6 shows the average energy consumed by each node running Feluda and VN2 for different network sizes. As the number of nodes increases, Feluda consumes much less energy than VN2. The reason is that every node running VN2 periodically generates two control packets (routing table information and protocol-specific counters) that travel a large number of hops. While traveling, the control packets may collide with other control or data packets, resulting in extra retransmissions. In contrast, Feluda does not use periodic transmission of out-of-band control packets. The only overhead stems from logging packet headers and query/response packet transmission. Please refer to [25] for the breakdown of average energy consumption in the 9×9 grid network. Due to flash memory reading/writing, Feluda consumes only 1∼2 mJ energy over the entire experiment, which is negligible compared to transmission/reception energy. In a 9×9 grid, Feluda consumes 57% less energy than VN2 on the average.

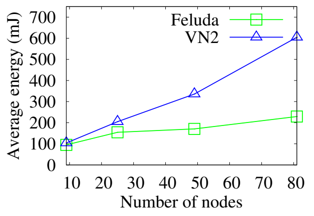


Fig. 6: Energy usage with varying grids (node down)

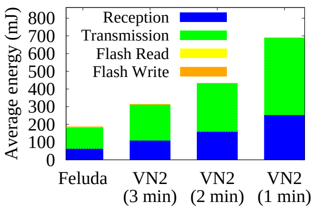


Fig. 7: Energy usage in 7×7 grid network (node reset)

We simulate four more failure types in a 9 × 9 grid network: congestion, incorrectly set node transmission power,

TABLE II: Energy usage in a 9×9 grid network for various failures

Failure type	Energy (mJ)	
	Feluda	VN2
Congestion	285.24	631.75
Erroneous TX power	266.91	619.49
Packet length error	273.11	627.15

TABLE III: ETX around the rebooted node

Tool	ETX
Feluda	147
VN2 (3 min)	27
VN2 (2 min)	32
VN2 (1 min)	112

data packet size exceeding the configured maximum value, and a forwarder node erroneously configured as a root. In order to introduce congestion in the network, we set the packet queue size of the data collection protocol to a small value (two) to induce packet drop. Feluda diagnoses congestion via the ACK flag in the packet header, as the flag indicates the reason for dropping a packet. We exploit the RSSI and length fields of packet headers to diagnose the cases where the transmission power of a node is changed to a high value causing link changes, and where the maximum packet length is configured to a small value causing transmission failure. Table II shows that Feluda reduces energy usage by at least 54.85% compared to VN2 in all these cases. Feluda diagnoses the erroneous root problem by checking headers of the routing ADVs (containing ETX of 0) from the false root. VN2 is unable to diagnose this problem, since it relies on the data collection protocol to transmit control packets, which are not forwarded toward the BS by the false root. By using a threshold of 0.5, Feluda effectively diagnoses these problems (and natural topological changes) within 55 s, whereas VN2 can take 3 minutes in the worst case.

2) Visibility Analysis: We conduct another experiment in a 7×7 grid network, where an arbitrary forwarder node reboots and immediately comes back up. In this case, descendants of this rebooted node experience packet loss and eventually change their parents. Once the rebooted node comes back, it starts broadcasting a routing ADV packet containing its initial ETX estimate towards the BS. The ETX value in this case remains high since the node has just started. It eventually converges to an accurate value after exchanging several rounds of routing ADVs with neighboring nodes. Hence, immediate reports from the neighboring nodes containing routing ADVs from the rebooted node can help diagnose the root cause of packet loss and link changes observed at the BS. Table III shows the ETX values reported by a neighbor of the rebooted node for Feluda and VN2 with different periodicity of control packet transmissions. Since Feluda collects information from the header of the routing ADV packet, it can report the correct ETX value broadcast by the faulty node just after its reset. In contrast, VN2 with 3-min periodicity reports an incorrect ETX value of 27. This affects the diagnosis accuracy. Clearly, VN2 accuracy can be increased by increasing the reporting frequency, but the energy consumption of VN2 increases. Fig. 7 shows average energy consumption for Feluda and VN2 with different periodicity. In this 7×7 grid network, Feluda consumes 73% lower energy compared to VN2 with a frequency of one minute.

3) Varying Node Distance: To understand the impact of link quality on energy usage, we consider a 5×5 grid with varying distance among nodes (10-40 m). We calculate energy usage and packet delivery rate (PDR) of Feluda and VN2. PDR indicates the percentage of packets delivered successfully to the BS. Fig. 8(a) shows that energy usage for both Feluda and VN2 increases as the distance among nodes increases. The reason is that the link qualities degrade as the distance increases, thereby causing more retransmissions in the network. Feluda consumes 32-48% less energy than VN2 when node distance exceeds 20 m. Fig. 8(b) shows that PDR for both data and control packets in VN2 decreases as the node distance increases. On top of poor link quality, collision between data and periodic control packets reduces the PDR. Note that energy usage for VN2 would have been higher if the data and control packets were not lost in the network and transmitted all the way towards BS. In contrast, Feluda does not use periodic control packets and maintains higher PDR at a lower energy cost.

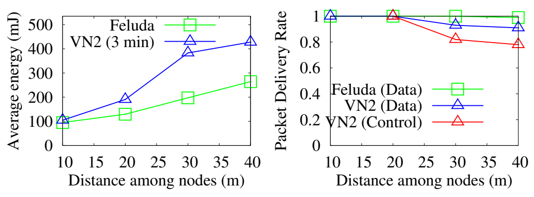
4) Linear Topology: We consider a linear topology which is widely used to monitor the health of infrastructure such as bridges and tall buildings. In our simulations, we vary the number of nodes in the linear topology from 10 to 25 and allow 30% of the nodes to generate data. Fig. 9 shows the energy consumption and PDR for Feluda and VN2 with varying numbers of nodes. As the number of nodes (*i.e.,* hops) increases, packets from distant nodes traverse longer paths, causing intermediate nodes to transmit and receive larger numbers of packets. Energy consumption for both Feluda and VN2 increases. Feluda reduces energy consumption by 28-31% compared to VN2. We find that packet reception rate decreases with increasing number of hops in the network as shown in Fig.9(b). The situation is worse for VN2 since both data and control packets suffer from loss due to queue overflow, thereby causing poor application performance and diagnosis accuracy. For example, the packet reception rate for data and control packets in VN2 becomes 64% and 47% respectively in a linear network of 25 nodes. In such networks with large hop counts, the energy gain with Feluda could have been higher if the data and control packets transmitted by VN2 had reached the BS.

B. *Testbed*

We construct a 6 m × 6 m topology of 25 battery-powered TelosB motes deployed in a 5×5 grid (details are given in [25]). Our motes have 10 kB RAM and 1 MB flash storage. We select transmission power level 2 to ensure multi-hop communication. The BS mote is placed in the corner of the testbed and connected to a laptop running the Feluda BS application via a serial interface.

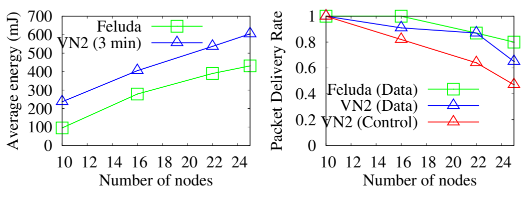
In our experiments, 30% of the motes actively generate data every 10 s. We set the sliding window size to 10 and failure detection threshold to 0.5 for Feluda. We consider two different periods of control packet transmissions for VN2: 1 min and 3 min. We use the same fault (node down) introduction strategy, performance metrics, and energy model as in Sec. V-A. Each experiment is 20 minutes long. During the experiment, every node records the number of bytes read from/written to flash, transmitted, and received into a local file every 30 s. These files are retrieved from all nodes at the end of the experiment to compute energy consumption.

Fig. 10 shows the average energy consumption per node for Feluda and VN2. VN2 periodic control packets incur energy cost. In contrast, Feluda stores packet headers into local flash



(a) Energy consumption (b) Packet delivery performance

Fig. 8: Energy usage and PDR vs. distance



(a) Energy consumption (b) Packet delivery performance

Fig. 9: Energy usage and PDR in linear network

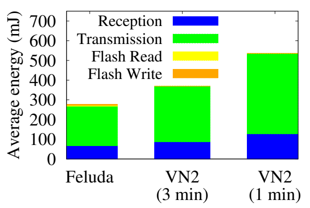


Fig. 10: Testbed results

memory and uses query/response packets only with potential network failures. The energy cost for flash read and write is negligible (3-5 mJ). Feluda reduces average energy usage per node by 24% over VN2 when it collects system metrics every 3 minutes. The energy reduction offered by Feluda becomes 48% when VN2 operates with 1 minute periodicity. These results validate the gain that we achieved in the COOJA simulations. Please refer to [25] for testbed experiments that demonstrate the effectiveness of Feluda in diagnosing topological changes caused by environmental conditions.

VI. Conclusions and Discussion

In this paper, we propose Feluda for exploiting packet headers and provenance to troubleshoot elusive network failures. The Feluda BS application tracks abnormal symptoms (*e.g.,* lost, delayed, or duplicate packets) and analyzes data provenance to localize failures. The Feluda code running on sensors allows them to efficiently store packet headers into flash and send headers of interest to the BS *on demand*. By requiring only a few nodes (in the vicinity of failures) to transmit a few packet headers, Feluda maintains packet-level visibility at a low energy cost. We implemented the Feluda BS application using Java and the sensor node module on ContikiOS. We evaluate Feluda through simulations and a testbed of 25 TelosB motes. Feluda identifies failures such as node down, soft reset, and congestion. We find that incorrectly configured nodes can induce routing changes, packet loss, and network partitions. Simulation and testbed results show that Feluda effectively diagnoses these failures with 24-57% lower energy cost than the state-of-the-art.

Feluda's centralized failure detection and on-demand querying methods keep the sensor nodes simple. Data collection applications configure the parameters of routing protocols running on sensor nodes. The query/response protocol within Feluda, however, runs independently of the data collection application and does not require run-time modification to the configuration of sensor nodes. Only the BS component of Feluda has configuration parameters. The query/response protocol is executed only when the Feluda BS application requires information about a specific packet (or node) to diagnose a detected failure. If the BS application can diagnose problems from node ID, sequence number, and provenance information embedded into packets, there is no overhead incurred by the query/response protocol. There is energy cost associated with storing packet headers into flash storage, but that is negligible as observed in the results. The energy reduction achieved by Feluda mainly stems from use of local logging and on-demand querying. Though advanced provenance embedding techniques such as PPF [7] reduce energy cost, our experiments consider the same method to embed node IDs for both Feluda and VN2 and demonstrate significant gains for Feluda.

With clustered failures (where a number of nodes fail simultaneously), Feluda can localize failures and determine the root cause as long as sufficient evidence is available. As future work, we plan to experiment with complex failures observed during long-term deployment of sensor networks and evaluate Feluda and state-of-the-art diagnostic protocols on large public testbeds.

References

[1] V. Sundaram, P. Eugster *et al.*, "Efficient diagnostic tracing for wireless sensor networks," in *ACM Sensys*, 2010.

[2] N. Ramanathan, K. Chang *et al.*, "Sympathy for the sensor network debugger," in *ACM Sensys*, 2005.

[3] X. Li, Q. Ma *et al.*, "Enhancing visibility of network performance in large-scale sensor networks," in *ICDCS*, 2014.

[4] K. Liu, Q. Ma *et al.*, "Self-diagnosis for large scale wireless sensor networks," in *IEEE INFOCOM*, 2011.

[5] Q. Ma, K. Liu *et al.*, "Sherlock is around: Detecting network failures with local evidence fusion," in *IEEE INFOCOM*, 2012.

[6] W. Zhou, Q. Fei *et al.*, "Secure network provenance," in *SOSP*, 2011.

[7] S. M. I. Alam and S. Fahmy, "A practical approach for provenance transmission in wireless sensor networks," *Ad Hoc Networks*, vol. 16, pp. 28 – 45, 2014.

[8] N. Handigol, B. Heller *et al.*, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *NSDI*, 2014.

[9] N. Tsiftes and A. Dunkels, "A database in every sensor," in *ACM Sensys*, 2011.

[10] N. Tsiftes, A. Dunkels *et al.*, "Enabling large-scale storage in sensor networks with the coffee file system," in *IPSN*, 2009.

[11] A. Dunkels, B. Gronvall *et al.*, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *IEEE LCN*, 2004.

[12] F. Osterlind, A. Dunkels *et al.*, "Cross-level sensor network simulation with COOJA," in *IEEE LCN*, 2006.

[13] X. Miao, K. Liu *et al.*, "Agnostic diagnosis: Discovering silent failures in wireless sensor networks," in *IEEE INFOCOM*, 2011.

[14] K. Liu, M. Li *et al.*, "Passive diagnosis for wireless sensor networks," in *ACM Sensys*, 2008.

[15] A.-R. M. Kamal, C. J. Bleakley *et al.*, "Failure detection in wireless sensor networks: A sequence-based dynamic approach," *ACM Trans. Sen. Netw.*, vol. 10, no. 2, Jan. 2014.

[16] B.-R. Chen, G. Peterson *et al.*, "LiveNet: Using passive monitoring to reconstruct sensor network dynamics," in *IEEE DCOSS*, 2008.

[17] M.-M. H. Khan, H. K. Le *et al.*, "Dustminer: Troubleshooting interactive complexity bugs in sensor networks," in *ACM Sensys*, 2008.

[18] J. Yang, M. L. Soffa *et al.*, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *ACM Sensys*, 2007.

[19] V. Sundaram and P. Eugster, "Lightweight message tracing for debugging wireless sensor networks," in *DSN*, 2013.

[20] K. Whitehouse, G. Tolle *et al.*, "Marionette: Using RPC for interactive development and debugging of wireless embedded networks," in *IPSN*, 2006.

[21] "Collection," http://www.tinyos.net/tinyos-2.x/doc/html/tep119.html.

[22] A. Dunkels, F. Österlind *et al.*, "An adaptive communication architecture for wireless sensor networks," in *ACM Sensys*, 2007.

[23] M. Keller, J. Beutel *et al.*, "Learning from sensor network data," in *ACM Sensys*, 2009.

[24] K. Lin and P. Levis, "Data discovery and dissemination with DIP," in *IPSN*, 2008.

[25] "Feluda: Provenance-enabled diagnosis of elusive network failures in wireless sensor networks," Tech. Rep., 2015, https://www.dropbox.com/s/gd78hy4g55znurx/techreport.pdf?dl=0.

[26] R. Martinez Oviedo, F. Ramos *et al.*, "A comparison of centralized and distributed monitoring architectures in the smart grid," *IEEE Systems Journal*, vol. 7, no. 4, pp. 832–844, Dec 2013.