Lightweight Message Tracing for Debugging Wireless Sensor Networks

Vinaitheerthan Sundaram*, Patrick Eugster†
*School of Electrical and Computer Engineering, †Department of Computer Science
Purdue University, West Lafayette IN 47907 USA
Email: {vsundar, peugster}@purdue.edu

***Abstract*—Wireless sensor networks (WSNs) deployments are subjected not infrequently to complex runtime failures that are difficult to diagnose. Alas, debugging techniques for traditional distributed systems are inapplicable because of extreme resource constraints in WSNs, and existing WSN-specific debugging solutions address either only specific types of failures, focus on individual nodes, or exhibit high overheads hampering their scalability.**

Message tracing is a core issue underlying the efficient and effective debugging of WSNs. We propose a message tracing solution which addresses key challenges in WSNs — besides stringent resource constraints, these include out-of-order message arrivals and message losses — while being streamlined for the common case of successful in-order message transmission. Our approach reduces energy overhead significantly (up to 95% and on average 59% smaller) compared to state-of-the-art message tracing approaches making use of Lamport clocks. We demonstrate the effectiveness of our approach through case studies of several complex faults in three well-known distributed protocols.

I. Introduction

Wireless sensor networks (WSNs) consist of many tiny, battery-powered sensor nodes equipped with wireless radios (a.k.a. motes) that sense the physical world and transmit the sensed information to a central "base station" computer via multi-hop wireless communication. Their small form factor and battery powered, wireless nature, makes WSNs suitable for a multitude of indoor and outdoor applications including environment monitoring (volcano [1], glacier [2]), structural monitoring (bridges [3]), border surveillance, and industrial machinery monitoring (datacenters [4]).

With WSNs being increasingly deployed to monitor physical phenomena in austere scientific, military, and industrial domains, runtime failures of various kinds are observed in many deployments [5], [1], [6]. In addition to node or link failures, failures engendered by complex interplay of software, infrastructure, and deployment constraints exhibiting as data races, timestamp overflows, transient link asymmetry, or lack of synchronization have been observed in distributed WSN protocols and applications [7], [8], [9]. Unexpected environmental factors arising from *in situ* deployment constitute the major cause for runtime failures occurring in WSNs despite careful design and validation. *Runtime* debugging tools constitute a promising approach to detect and diagnose generic runtime failures in WSNs.

Runtime debugging is a challenging problem even in "traditional" resource-rich wireline networks. While *online* debugging techniques [10], [11] are useful in reducing the latency of fault detection and diagnosis, they tend to incur high runtime overhead and are susceptible to Heisenbugs (faults that disappear when the system is observed). *Offline* debugging techniques [12], [13], [14], [15], [16], [17], [18] are inapplicable in WSNs as large amounts of data memory and non-volatile memory are required to store megabytes of traces generated for subsequent offline mining.

WSN-specific online debugging approaches [19], [20], [21], [22], [8] focus on providing visibility into the network as well as remote control of it. While these approaches are very useful for small-scale testbeds, they are not suitable for debugging after deployment, as they are energy-inefficient and are highly susceptible to Heisenbugs due to non-trivial intrusion via computation and communication overheads. Several WSN-specific trace-based offline debugging techniques have thus been proposed [23], [24], [25], [26], [27], [9], [28]. Some of these solutions [23], [24] focus on coarse-grained diagnosis, where the diagnosis pinpoints a faulty node or link, or network partition. Some approaches achieve automation [28] but require multiple reproduction of failures to learn the correct behavior with machine learning techniques. Some other approaches [26], [27], [9] focus on node-level deadlocks or data races. While these offline solutions are useful for diagnosing various runtime failures, they do not support generic, resource-friendly *distributed* diagnosis of complex failures occurring through sensor node *interaction*, in end applications as well as core protocols.

Tracing of message sends and receives is a cornerstone of distributed diagnostic tracing. To faithfully trace distributed program behavior, it is of utmost importance to be able to accurately pair message sends (cause) and receives (effects). When observing distributed failures in WSNs (see Section IV) four specific key constraints for a generic and efficient message tracing solution emerge:

1) Resource constraints. As stated, WSNs are highly resource-constrained, and thus mechanisms for tracing distributed interaction via message sends and receives must impose only overheads. In particular, traces on individual nodes should be well compressible to reduce storage and communication overheads, which dominate the tracing overhead [9], [27].

2) Message losses. Pairing of message sends and receives can not simply be inferred from *sequences* of such events, when individual messages can get lost. Yet, due to the *inherent dynamic* nature of WSNs, best-effort transmission protocols are commonly used directly.
3) Out-of-order reception. Similarly, basic communication protocols used directly by end applications and other protocols do not provide ordered message delivery, which adds to the difficulty of pairing up message sends and receives.
4) Local purging. When trace storage is full, the decision to rewrite the trace storage has to be a local decision for energy efficiency purposes. Since external flash is very limited (about 512KB - 1MB), the traces fill the storage quickly. Purging traces locally at arbitrary points of the execution complicates pairing up message sends and receives.

The state-of-the-art fails in some requirements (see Section II-B). This holds in particular for the golden standard — originating from wireline networks [16], [29], [18], [13], [30], [14] and then adapted for WSNs [31] — consisting in identifying messages with Lamport clocks [32] paired with sender identifiers: besides generating false positives at replay (Lamport clocks being *complete* but not *accurate*) this solution does not inherently support losses and is not as lightweight as it may seem at first glance.

In this paper, we propose a novel message tracing scheme for WSNs that satisfies all the four requirements above. Our approach exploits restricted communication patterns occurring in WSNs and consists of three key ideas: (1) use of per-channel sequence numbers, which enables postmortem analysis to recover original ordering despite message losses and out-of-order message arrivals, (2) address aliasing, where each node maintains a smaller id for other nodes it communicates more often with, and (3) optimization for the common case of in-order reliable delivery. We combine our message traces with the local control-flow trace of all events generated by the state-of-the-art [9], [27] to get the entire trace of the distributed system.

Specifically, this paper makes the following contributions:

- We present a novel distributed message tracing technique that satisfies our four constraints.
- We show the effectiveness of the distributed traces achieved by our message tracing technique in combination with control-flow path encoding for individual sensor nodes with the open-source TinyTracer [9] framework via several real-world WSNs distributed protocol faults described in the literature.
- We analytically demonstrate the significant reduction in trace size and empirically demonstrate the ensuing energy savings (up to 95% and on average 59%) of our technique over the state-of-the-art, irrespective of its inconsistencies in tracing of communication (and thus misdiagnosis) in the presence of message losses and out-of-order message arrivals.

Section II defines the problem with necessary background. Section III presents the design and implementation of our approach. Section IV shows the effectiveness of our approach through case studies of several common distributed protocols in WSNs. Section V shows space efficiency analytically. Section VI shows energy efficiency empirically. Section VII discusses related work and Section VIII concludes the paper.

II. Background and Problem Definition

In this section, we first describe the existing approaches in trace-based debugging. We then relate these to the challenges specific to WSNs and identify key requirements for tracing to be useful for distributed faults diagnosis in WSNs.

A. Trace-based Debugging of Distributed Systems

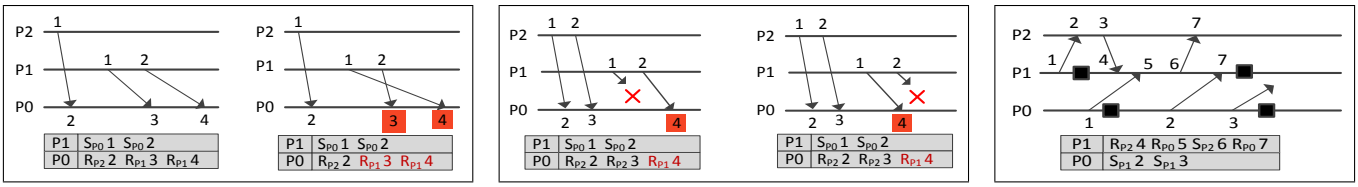
Trace-based replay debugging [16], [18] is a promising approach for debugging distributed systems [12]. A correct replay is one in which the causal ordering of messages observed in the original execution is maintained. Causal ordering of messages is defined as follows. A message send causally precedes its corresponding receive, and any subsequent sends by the same process. If a message m_1 received by a node before it sends another message m_2, then m_1 causally precedes message m_2. Causal ordering is transitive, i.e., if m_1 causally precedes m_2 and m_2 causally precedes m_3, then m_1 causally precedes m_3.

To obtain the causal ordering of the original execution, the message dependences have to be recorded in the trace. In trace-based replay solutions [33], [29] for wired distributed systems, the message dependences are captured using logical clocks [32], [34]. Originally proposed for enforcing ordering of events (including messages) in fundamental distributed systems problems such as ordered broadcast and mutual exclusion, logical clocks are used here to capture the ordering during the original execution and recreate it in the replay.

Lamport clocks [32] use a single integer maintained by each node. While scalable, they are inaccurate meaning some concurrent events are classified as causally related. This inaccuracy can slow down replay of a network because concurrent events can be replayed in parallel threads, and yield false positives. To overcome inaccuracy, *vector clocks* [34] can be used which precisely capture concurrent and causally related events. Vector clocks have been used to identify racing messages and by recording only those racing messages, trace sizes can be reduced considerably [35]. Since vector clocks maintain n integers, where n is the number of nodes in the network they impose high overhead and do not scale well. Between these two extremes of Lamport clocks and vector clocks, there are other logical clocks such as *plausible clocks* [36] or *hierarchical clocks* [37]. However, most tracing-based replay solutions [33], [29] use Lamport clocks because of their ease of implementation and scalability.

B. Inapplicability of Existing Approaches

Existing trace-based replay solutions for *wired* distributed systems work under the assumptions of abundance of energy



(a) Out-of-order message arrivals (b) Message loss (c) Local purging

Fig. 1. Examples showing the shortcomings of Lamport-clock based message tracing when pairing message receive events with the corresponding send events in the presence of unreliable channels or arbitrary local purging of traces. The traces from processes $P0$ and $P1$ are shown below the space-time representation of processes. Figure 1(a) shows a simple example with a same trace being generated in both cases where messages arrive in order and out of order. It is impossible to correctly identify out-of-order arrivals from the trace during post-mortem analysis, implying that message receive events cannot be paired with corresponding sends. Figure 1(b) shows an example in which it is impossible to identify which message was lost from the trace during postmortem analysis. Figure 1(c) shows an example in which it is impossible to tell whether the receive event 7 in process $P1$ pairs with send event 2 or 3 in process $P0$ just by looking at the traces. The square black dots represent the points in time where the local traces are purged.

(connected to wall-socket), storage in the order of gigabytes, and network bandwidth in the order of at least kilobytes per second. More importantly, these distributed applications are assumed to run on top of a FIFO reliable communication layer such as TCP. These assumptions do not hold in WSNs and any WSN message tracing solution should cope with (1) stringent resource constraints, (2) out-of-order message arrivals, (3) message losses, and (4) local purging. Furthermore, the resource constraints in WSNs requires the traces recorded to be highly compressible, which means newly recorded information has less variability from previously recorded information. We show that the existing approaches cannot cope with unreliability and local purging as well as are not very compressible.

The existing approaches that record logical clocks alone cannot recreate the causal order correctly in the presence of unreliability. This is true even for approaches using vector clocks [38]. Combining Lamport clocks with sender addresses as proposed by Shea [31] can still lead to inconsistent causal ordering due to unreliable communication. We show this in the case for Lamport clocks with the help of counter-examples.

Figure 1 shows the counter-examples as a *s*pace-time diagram representation of processes and their message interaction. The horizontal lines in the space-time diagram represent the processes with time increasing from left to right and the arrows represent messages, with the direction of arrow from sender to receiver. The traces contain the event type (send/receive), process identifier, and the Lamport clock value. Such traces cannot correctly pair up message send events with their corresponding receive events when there are out-of-order message arrivals, message losses or arbitrary local purging of traces.

Figure 1(a) illustrates that the traces cannot correctly pair up message send events with their corresponding receive events when the underlying channel can reorder messages. In this example, it is not possible to pair the receive events 3 and 4 in process $P0$ with the corresponding send events.

Figure 1(b) illustrates that the traces cannot identify the message send event corresponding to a lost message. In this example, it is impossible to pair the receive event 4 in process $P0$ with the corresponding send event.

Figure 1(c) illustrates that the traces cannot correctly pair up message send events with their corresponding receive events when the traces are purged locally to handle full trace buffers. The black square dots represent the points in time when the traces are locally purged. The traces shown for process $P0$ and process $P1$ are the snapshots of the respective trace buffers. In this example, either of the receive events 5 and 7 in process $P1$ pairs with send events 2 and 3 in process $P0$ and so it is impossible to pair receive event 7 in process $P1$ with the corresponding send event 2 in process $P0$.

In the cases discussed above, the problem is that Lamport clocks count events in the distributed system *globally*, which means the clock value depends on multiple nodes. Such global counting causes logical clocks to increase their values without regular intervals, which reduces the opportunities for compression.

C. Tracing Concurrent Interprocedural Control-flow

CADeT can be combined with node-local approaches such as [27], [9] to diagnose distributed faults. We summarize one such state-of-the-art local tracing approach called TinyTracer [9] as we use it in our evaluation. TinyTracer encodes the interprocedural control-flow of concurrent events in a WSN before discussing fault case studies. First, let us consider the intraprocedural encoding. If there are n acyclic control-flow paths in a procedure, it can be encoded optimally with $log\ n$ bits as an integer from 0 to $n-1$. In their seminal paper, Ball and Larus [39] proposed an algorithm that uses minimal instrumentation of the procedure to generate the optimal encoding at runtime. TinyTracer extends that approach to generate interprocedural path encoding of all concurrent events in WSN applications written in nesC for TinyOS, a widely used WSN operating system. The technique records the event identifier at the beginning of the event handler and the encoding of the interprocedural path taken inside the event handler and an end symbol at the end of the event handler. The trace generated by the approach would include all the concurrent events along with the interprocedural path taken in the order the events occurred.

D. Problem Definition

Thus in this paper, we address the problem of *how to enhance local control-flow traces such that distributed faults in WSNs can be diagnosed efficiently?*

III. CADET

We propose a novel efficient decentralized compression-aware message tracing technique that records message order correctly and satisfies the WSN specific requirements.

A. Design Rationale

We exploit the following WSN application characteristics.

1) Nodes most commonly communicate with only few other nodes, usually the neighbors or special nodes such as cluster heads or a base station. [40], [41]
2) Nodes local control-flow trace can be used to infer the contents of the message such as type and local ordering. [9].
3) The common case is that messages are not lost and arrive in order, though such incidents must be handled.

The key idea of our design is to optimize for the common case, i.e., when there are no message losses or out-of-order message arrivals. For the common case, we record minimal information required to trace a message and ensure that information is compressible, which means the recorded information for a message has less variability from previously recorded information. This is achieved by maintaining some in-memory state which is periodically recorded into the trace and serves as local checkpoint. When a message loss or out-of-order message arrival occurs, we store additional information to infer it.

There are many advantages of our design. First, our design allows message sends and receives to be paired for both unicast and broadcast even in the presence of unreliability. Second, our design is compression-aware, i.e., it records information such that it can be easily compressed. Third, our design allows lightweight local checkpointing and the checkpoints store information about the number of messages sent/received with every node it communicates with. Fourth, our design is efficient because it uses only one byte sequence numbers as the sequence numbers are unique to each *pair* of nodes and take long time to wrap around.

B. Data Structures

Compression-aware distributed tracing hinges on two key techniques, namely, *address aliasing* and *per-partner sequence numbers*. We refer to the nodes that communicate with a particular node as *partners* of that node. For each partner, a local alias, which can be encoded in fewer bits compared to the original address (unique network address), is assigned when a communication is initiated or received from that partner. This mapping (one-to-one) from original addresses to aliases is maintained in an address alias map, AAMap. For each local alias, we also maintain a pair (last sequence number sent, last sequence number received) in a partner communication map, PCMap.

C. Algorithm Description

Algorithm 1 presents our message tracing algorithm.

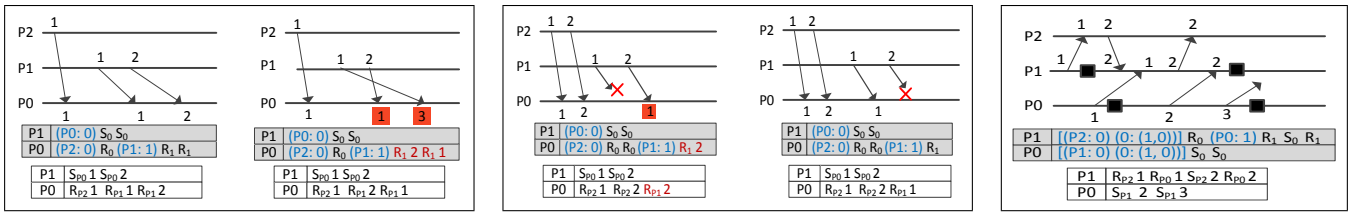
When a node Q sends a message to a partner P the sender address Q and the next sequence number are appended to the message and *the send event and the alias for* P are recorded in the trace. Observe that the next sequence number is *not* recorded. We update the PCMap with the new sequence number. Suppose the partner is not present in AAMap, then an alias for that partner is added to AAMap and the new alias with pair (0,0) is added to PCMap.

When a node P receives a message from a partner Q the sequence number received in the message is checked against the expected sequence number (= last sequence number received + 1) from the PCMap at P. If the sequence number in the message is the same as the expected sequence number, then the *receive event and the alias of* Q are recorded in the trace. The PCMap is updated with the expected sequence number

Algorithm 1 Tracing message sends. AAMap is a map from partner addresses to local aliases and PCMap a map from local aliases to respective communication histories. LOOKUPAAMAP returns the unicast alias of the message network address of the destination. If the destination address is not present, the destination address is added along with the next available local alias to the AAMap and that alias is returned. Similarly, LOOKUPAAMAPBCAST returns the broadcast alias for the network address, which is different from the unicast alias. LOOKUPPCMAP returns the communication history of the partner. If the partner alias is not present, it is added along with the pair (0,0) to the PCMap and the pair (0,0) is returned indicating no communication history in the map. The address of the node, the message's destination address and the message's source address are respectively shown as `myAddr`, `msg.destAddr`, and `msg.sourceAddr`.

```
1: UPON SEND (msg)
2: if msg.destAddr is not a broadcast address then
3:     alias ← LOOKUPAAMAP (msg.destAddr)
4: else
5:     alias ← LOOKUPAAMAPBCAST (myAddr)
6: end if
7: (lastSentSeq, lastRcvdSeq) ← LOOKUPPCMAP (alias)
8: nextSendSeq ← lastSendSeq + 1
9: APPENDTOMESSAGE (myAddr, nextSendSeq)
10: RECORDTOTRACE ('S', alias)
11: UPDATEPCMAP (alias, (nextSendSeq, lastRcvdSeq))

1: UPON RECEIVE (msg)
2: if msg.destAddr is not a broadcast address then
3:     alias ← LOOKUPAAMAP (msg.sourceAddr)
4: else
5:     alias ← LOOKUPAAMAPBCAST (msg.sourceAddr)
6: end if
7: (lastSentSeq, lastRcvdSeq) ← LOOKUPPCMAP (alias)
8: expectSeq ← lastRcvdSeq + 1
9: if msg.seq = expectSeq then
10:     UPDATEPCMAP (alias, (lastSendSeq, expectSeq))
11:     RECORDTOTRACE ('R', alias)
12: else if msg.seq > expectSeq then
13:     UPDATEPCMAP (alias, (lastSendSeq, msg.seq))
14:     RECORDTOTRACE ('R', alias, msg.seq)
15: else
16:     RECORDTOTRACE ('R', alias, msg.seq)
17: end if
```



(a) Handling out-of-order message arrivals (b) Handling message loss (c) Handling trace chunks

Fig. 2. Traces generated by CADeT for the same set of processes and messages in Fig. 1. The red colored entries in trace shows that message loss and out-of-order message arrivals can be distinguished correctly and message sends and receives can be paired correctly. In Figure 2(c) the local checkpoints are shown as black square dots on the space-time diagram. The internal data structures are not shown.

indicating that the sequence number has been successfully received. This is the common case when there are no message losses or out-of-order message arrivals.

If the sequence number in the message is greater than expected (some message loss or out-of-order message arrival happened), then PCMap is updated with the sequence number of the message as the last sequence number received. If the sequence number in the message is less than expected (some old message is arriving late), the PCMap is *not* updated. In both the cases of unexpected arrivals, the receive event, the alias of Q, and *the unexpected sequence number* are recorded in the trace. Note that we record the unexpected sequence number information in the trace to correctly pair messages in the case of message loss or out-of-order message arrival.

In WSNs, broadcasts to neighbors are not uncommon – e.g., advertise detection of an intruder to your neighbors. It's necessary to handle broadcast to be able to pair sends and receives correctly. To handle broadcast, we treat each node to have two addresses, its own address and its own address with a broadcast marker. Thus, when a broadcast is sent or received, it is counted separately from the unicast. For example, when a node P sends a broadcast followed by a unicast to node Q and assuming no other communication happened in the network, node Q's AAMap map will have two entries, one for node P (unicast receive), and another for node $P*$ (broadcast receive) and it's PCMap will contain two $(0, 1)$ entries corresponding to the two messages received from node P. Similarly, node P will have two entries in AAMap corresponding to node Q (unicast send) and node $P*$ (broadcast send) and it's PCMap will contain two $(1, 0)$ entries corresponding to the two sends. Both the unicast send and the broadcast send of node P can be correctly paired with their corresponding receives at node Q as the send and recieve events are counted separately.

Figures 2 shows how CADeT handles out-of-order message arrivals, message losses and local purging for the scenarios shown in Figures 1.

D. Proof Sketch

We informally argue the correctness of the algorithm. The goal of the algorithm is to track the order of message receptions. Because sender-receiver pairs are handled independently of each other by including sender identifiers, it is sufficient to consider one sender-receiver pair. Assume a sequence of messages *received* with respective sequence numbers [i_1, i_2, i_3, i_4,..]. Rather than logging the numbers, an equivalent way is to log the first, then the differences i_1, [$i_2 - i_1$, $i_3 - i_2$, $i_4 - i_3$, ...]. The original order can be trivially reconstructed. The numbers in the original sequence need not be ordered which supports out-of-order message reception and message losses. In our case, the difference between adjacent numbers in the sequence is commonly 1, which can be exploited by logging a simple predefined tag rather than the difference value. Otherwise we log the number itself which is equivalent to logging the difference as explained above. Since senders use monotonically increasing per-receiver counters the differences between subsequent message *sends* are invariably 1 and sequence numbers are unique, allowing for correct pairing. Since broadcast uses separate counters, the same proof applies.

IV. Distributed Fault Case Studies

With the help of real-world bug case studies, we show that the distributed control-flow traces generated by CADeT together with TinyTracer [9] aid in diagnosing complex faults in distributed protocols proposed for WSNs. First, we present LEACH [42], a WSN clustering protocol, followed by diagnosis of two faults diagnosed in its implementation. Next, we present diagnosis of faults in WSNs designed as pursuer-evader networks [8]. Finally, we present diagnosis of two practical issues in directed diffusion [7], a scalable and robust communication paradigm for data collection in WSNs. In all the case studies, we assume the presence of CADeT's trace of messages as well as trace of message send and receive events local control-flow.

A. LEACH

LEACH [42] is a TDMA-based dynamic clustering protocol. The protocol runs in rounds. A round consists of a set of TDMA slots. At the beginning of each round, nodes arrange themselves in clusters and one node in the cluster acts as a cluster head for a round. For the rest of the round, the nodes communicate with the base station through their cluster head. The cluster formation protocol works as follows. At the beginning of the round, each node elects itself as a cluster head with some probability. If a node is a cluster head, it sends an advertisement message out in the next slot. The nodes that are not cluster heads on receiving the advertisement messages from multiple nodes, choose the node closest to them based on the received signal strength as their cluster head and

send a join message to that chosen node in the next slot. The cluster head, on receiving the join message, sends a TDMA schedule message which contains slot allocation information for the rest of the round, to the nodes within its cluster. The cluster formation is complete and the nodes use their TDMA slots to send messages to the base station via the cluster head.

B. Network Congestion in LEACH

1) Fault Description: When we increased the number of nodes in our simulation to 100, we found that data rate received at the base station reduced significantly. The nodes entered `NO-TDMA-STATE` and didn't participate in sending data to the clusterhead. The reason was that many nodes were trying to join a cluster in the same time slot. Due to the small size of the time slot, Join messages were colliding. Consequently, only fewer nodes successfully joined clusters. The nodes that did not join the cluster in a round remained in `NO-TDMA-STATE` resulting in lower throughput. To repair the fault, we increased the number of time slots for TDMA and introduced a random exponential backoff mechanisms.

2) Diagnosis with CADeT: When the throughput dropped at the base station, the traces from several nodes were examined. The abnormal control-flow in the trace revealed that some nodes did not have a slot assignment. We then confirmed that TDMA schedule broadcast was indeed received. We analyzed the trace to find the cluster head from the Join message sent to the cluster head in that round. When trying to pair the Join messages, we noticed the cluster head did not receive the Join message and therefore, did not allocate a slot for that node in the TDMA schedule. Since Join message send was recorded but not receipt, the link between cluster node and the cluster head must have failed either due to congestion or channel corruption. When we made the channels perfect in our simulations, we still observed the same result leading us to identify Join message collision as only possible explanation for link failure.

C. Data Race in LEACH

1) Fault Description: When we increased the number of nodes in the simulation to 100, we noticed significant reduction in throughput. Similar to the above case study, the nodes entered `NO-TDMA-STATE` and didn't participate in sending data to the clusterhead. However, the root cause was different. The problem was due to a data race between two message sends that happens only at high load.

After sending the TDMA schedule message, the cluster head moves into the next state and sends a debug message to the base-station indicating it is the cluster head and the nodes in its cluster. When the load is high, the sending of TDMA schedule message may be delayed because of channel contention. This in turn affects the sending of debug message as the radio is busy. In WSNs, message buffers are usually shared among multiple sends. It's not uncommon to use one global shared buffer for sending a message as only one message can be sent at a time. When attempting to send the debug message, before checking the radio was busy, the message type of the global shared buffer was modified unintentionally and therefore, the TDMA schedule message was modified into a debug message. Due to this implementation fault, the global send buffer was corrupted which resulted in wrong message being delivered. The nodes in the cluster dropped this message after seeing the type, which is intended only for the base station. This error manifested only when the number of nodes was increased because the increase in load caused the TDMA schedule message to be retried several times and the original time slot was not enough for the message transmission. We fixed this error by removing the fault as well increasing the time slot size to send TDMA message.

2) Diagnosis with CADeT: We examined several node traces after noticing poor throughput. We found that the cluster nodes were in the same state `NO-TDMA-STATE` as the above case study. Since we fixed the join message congestion, we examined the traces closely and noticed that some unexpected message was received after sending the Join message. When we paired that message receive with the sender, we realized that message was a TDMA schedule message. From the receiver trace control-flow, it was clear that the message was of unexpected type. However, the message was not garbled as it passed the CRC check at the receiver. This indicated that the problem was at the sender. We examined the sender's control-flow closely and *the trace indicated that there was a state transition timer event fired between the TDMA schedule message send and the corresponding* `sendDone` *event in the cluster head.* From the sender's control-flow, we noticed that debug message send interfered with the TDMA send and the implementation fault that corrupted the message buffer was discovered.

D. Intrusion Detection Failure in Pursuer-Evader Networks

WSNs used for military or border surveillance [43], [44], [45] are modeled as pursuer-evader games, where the WSN is the pursuer and the intruder is the evader. The main goal of these WSNs is to alert the base station when an an intruder is detected by sensors. To avoid congestion of alerts sent to the base station, one node acts as a leader and alerts the base station of the intruder. The following simple decentralized leader election protocol is employed. The nodes broadcast the signal strength detected to their neighbors and the node with the strongest signal elects itself as the leader. In these WSNs, failing to detect an intruder is a serious problem and hence needs to be diagnosed.

1) Fault Description: The failure to detect an intrusion can be caused by link asymmetry, time synchronization error [8], or link failure. Let node *A* be the node with the strongest signal during an intrusion. If there is link asymmetry, node *A* would not get neighbors broadcast while they get node *A*'s broadcast. The neighbors would assume node *A* will elect itself as leader. However, node *A* would falsely assume that the signal detected locally was spurious because it did not hear from other neighbors. Therefore, node *A* will not elect itself as the leader and the intrusion will not be detected. A similar situation may arise if there is a time synchronization error.

Node *A* may check for neighbors broadcast before they are supposed to be received because of time synchronization error. Node *A* would falsely assume spurious local detection and not elect itself as a leader. If the link between node *A* and the base station fails, the intrusion detection failure occurs. In addition, intrusion detection failure can occur due to implementation fault in the code. It is important to detect, diagnose and repair such failures. Missing an intrusion can be determined by the base station if the intruder gains illegal access or some other part of the network catches the intruder.

2) Diagnosis with CADeT: When a failure report is received at the base station, it pulls the recent traces from the neighborhood. Note that the traces may contain messages exchanged before and after the intrusion because these WSNs are constantly running. Since CADeT traces allow the ability to pair message sends and receives despite losses, it is possible to identify the time window in which the intrusion occurred (when there was broadcast among neighbors). Note that even when there are multiple intrusions, each intrusion occurrence can be identified due to increasing sequence numbers assigned to the message exchange generated by the intrusion. When the time window of the intrusion is identified from the traces, the detection failure can be narrowed down. If the traces show that an alert was sent by a node but that alert was not received at the base station, then the reason is the failure of link between the elected leader and the base station. If the traces show that a node, say node *A* has not recorded local broadcasts receipts but other nodes traces reveal the local broadcast sends and receipts, it is likely this node suffers from link asymmetry. If the control flow of other nodes show that those nodes did not expect to become the leader, then it is clear that this node was supposed to be the leader but due to link asymmetry it did not become a leader. If the node *A*'s trace does have the receipt of the broadcast messages but the control-flow shows that node *A* assumed that the local detection of the intruder was a spurious signal before receiving the broadcast messages, it is likely node *A* was the supposed-to-be leader that was unsynchronized with the other nodes.

E. Serial Message Loss in Directed Diffusion

Directed diffusion [46] is a communication paradigm that allow nodes (sink nodes) to express interest in data from other nodes in the network and these interest messages are propagated throughout the network through controlled flooding. The nodes matching the interest act as source nodes and send data back to the sink nodes using the paths taken by interest but in the opposite direction. An interest can be satisfied by a single data message or a stream of data messages from the source nodes. To achieve directed diffusion, nodes maintain an interest cache and a data cache. When an interest message is received, a node adds an entry to its interest cache if it is not already there, forwards the interest message to its neighbors other than the interest message sender, and creates a gradient (parent) towards the neighbor that sent the interest message. When a data message is received, a node checks for a matching interest in the interest cache. If a matching interest is present and the data message is not in the data cache, the data message is added to the data cache and is forwarded to all parents that have expressed interest in that data. Once the data flow for an interest is stabilized, the interest message will be renewed only through most reliable neighbor, and thereby, reducing duplicate traffic eventually.

1) Fault Description: There are two practical issues, namely, timestamp overflow and node reboots that are not handled well in directed diffusion design and both issues manifest as a continuous loss of messages at a different node (parent node) as discussed in Khan et al. [7]. Let node *A* be a source or forwarding node that satisfies an interest from the parent node *B*. In the case of timestamp overflow, parent node *B* drops the packet because of older timestamp. However in the case of node *A* reboot, node *A* drops the packet to be sent/forwarded to parent node *B* because its interest cache is wiped out after reboot. In both cases node, the manifestation is the same, which is *B* observes lower message rate and continuous message loss.

2) Diagnosis with CADeT: When node *B* reports loss of messages from node *A*, trace from these nodes are pulled to the base station. The last message sent from node *A* to node *B* is compared with the last message received at node *B* from node *A*. If they two match, it implies node *A* has not been sending more messages and perhaps has some problems. A mismatch implies the messages are either dropped at node *B* or link failure. In the former case, node *A*'s control-flow trace is examined, which would reveal a reboot as the initialization functions called after the reboot would appear in the control-flow trace. In the latter case, node *B*'s control-flow trace is examined. If the timestamp overflow happened, the control-flow trace would show that the code took a different path at the condition checking timestamp of the messages. If none of the two cases happened, the message loss is the most likely due to failure of link between nodes *A* and *B*.

V. Analytical Evaluation

We analytically compare our approach to the adapted state-of-the-art, Liblog [29], showing that our approach reduces the trace size. For tractability and fairness we made several simplifying assumptions. First, traces are uncompressed. Second, each node communicates with a small subset of nodes called its partners. Among its partners, each node communicates with some of them regularly and some of them irregularly. Third, each node sends messages at the same rate to its partners. The notation used in the analysis is shown in Table I.

A. CADeT

The size of the trace generated at a node by CADeT (γ_N) depends on the checkpoint stored and the trace entries generated by messages sent and received by that node.

First, we calculate the size of the trace generated at a node in a checkpoint interval due to messages sent by a node (γ_S). The number of messages sent in a checkpoint interval, α, is the product of average message rate from a node to a partner

TABLE I
NOTATION USED IN THE ANALYSIS

n	number of nodes in a WSN
n_P	number of partners that a node communicates with in a checkpoint interval
n_R, n_O	number of partners that a node communicates regularly and occasionally respectively
t_C	checkpoint interval
r_S	rate of messages sent from a node to a partner
p_O	percentage of occasional partners that a node communicates with in a checkpoint interval
p_I	probability that a message arrives in order
p_L	probability of a message loss
b_I	number of bytes required to record a message send or an in-order message receive
b_S	number of bytes required to record a sequence numbers by CADeT
b_A, b_P	number of bytes required to record an AAMap entry and a PCMap entry in the trace respectively
b_F	number of bytes required to record a sequence number by Liblog
γ_N	size of the trace generated by CADeT due to messages sent to and received from all its partners in a checkpoint interval
γ_S, γ_R	the size of the trace generated by CADeT due to messages sent to a partner or received from a partner respectively in a checkpoint interval
γ_C	size of the trace generated by CADeT due to recording a checkpoint in the trace
δ_N	size of the trace generated by Liblog due to messages sent to and received from all its partners in a checkpoint interval
δ_S, δ_R	size of the trace generated by Liblog due to messages sent to a partner or received from a partner respectively in a checkpoint interval

(r_S) and the total length of the checkpoint interval (t_C) and is given in Eq. 1.

$$\alpha = r_S \, t_C \tag{1}$$

The number of bits used to represent each message in the trace (b_I) is the sum of number of bits used to record that a message has been sent and the number of bits to record the destination alias. Since there are α messages sent in a checkpoint interval and each message generates b_I bits of trace, the size of the trace generated at a node in a checkpoint interval due to messages sent by a node (γ_S) is their product and is given in Eq. 2.

$$\gamma_S = \alpha \, b_I \tag{2}$$

Next, we calculate the size of the trace generated at a node in a checkpoint interval due to messages received from another node, γ_R, as follows. Out of the messages sent to a node in a checkpoint interval, α, only the fraction $(1 - p_L)$ of them are received and lead to a trace entry, where p_L is the probability of a message being lost. In CADeT, the size of the trace entry of a received message depends on whether a message was received in the order expected or not. If a message arrives when it was expected, then the receive event 'R' and the sender alias are recorded in the trace and this requires b_I bytes, similar to recording a message send. However, if a message arrives earlier or later than expected, then CADeT stores the sequence number of the message into the trace in addition to the b_I bytes recorded in the trace. If the probability of a message arriving when expected is p_I, γ_R is given in Eq. 3.

$$\gamma_R = (1 - p_L)\left((\alpha \, b_I \, p_I) + (\alpha \, (b_I + b_S)\,(1 - p_I))\right) \tag{3}$$

$$= (1 - p_L)\,(\gamma_S + \alpha \, b_S (1 - p_I)) \tag{4}$$

In addition to tracing message sends and receives, CADeT dumps the internal tables (AAMap and PCMap) to the trace once every checkpoint interval. This is referred to as the checkpoint. Since the table contains one entry per partner, we need to estimate the number of partners a node communicates with in a checkpoint interval.

Each node communicates with its n_R nodes regularly called its regular partners and n_O nodes occasionally, called its occasional partners, over its lifetime. We observe that the number of partners a node communicates with over its lifetime, given by $n_R + n_O$, is much less than the total number of nodes in the network (n).

In a checkpoint interval, we assume that a node communicates with all its regular partners (n_R) and some of its occasional partners ($n_O \quad p_O$), where p_O is the percentage of occasional partners a node communicates with in a checkpoint interval. Now, the number of partners a node communicated with in a checkpoint interval (n_P) is given by Eq. 5.

$$n_P = n_R + n_O \, p_O \tag{5}$$

The size of that checkpoint, γ_C, is the product of the size of the internal table entries, b_A and the number of partners this node communicated within that interval.

$$\gamma_C = (b_A + b_P)\, n_P \tag{6}$$

Since each node communicates with n_P partners in a checkpoint interval, the trace size due to messages sent and received by a node is $(\gamma_S + \gamma_R)\, n_P$. Thus, γ_N is given by Eq. 7.

$$\gamma_N = \gamma_C + (\gamma_S + \gamma_R)\, n_P \tag{7}$$

$$= n_P \left((b_A + b_P) + \gamma_S \,(2 - p_L) + \alpha \, b_S \,(1 - p_I)\,(1 - p_L)\right) \tag{8}$$

B. Liblog

A common way to capture message interactions in a distributed systems for replay is to record the Lamport clock along with local non-determinism, including capturing entire messages [29], [18], [33]. Since recording a full message is prohibitively expensive, we adapt Liblog [29] for WSNs as follows: record the sender address along with the Lamport clock for every message. This was also suggested by Shea [31]. Henceforth, Liblog refers to this adapted version of Liblog. Although it may lead to inconsistent replay when message losses or out-of-order message arrivals can happen as described before (cf. II-B), we use Liblog because it is the known state-of-the-art.

We calculate the size of the trace generated by Liblog in a checkpoint interval, δ_N, similar to CADeT. Since Liblog records Lamport clocks in the trace entry, the periodic checkpoints taken by CADeT are not required as the clock value is cumulative. However, the number of bits to record the Lamport's clock, b_F is larger as Lamport's clock increases with each message sent from all the nodes. Therefore, the size

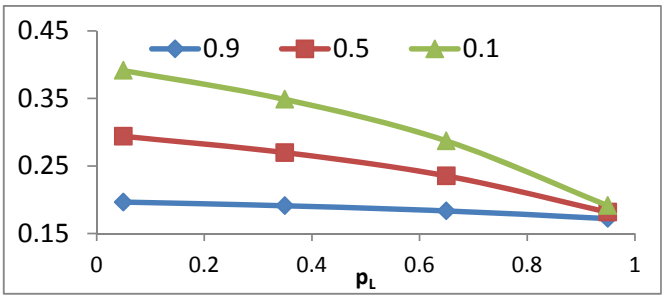


Fig. 3. Effect of probability of loss and out-of-order message arrivals on $\frac{\gamma_N}{\delta_N}$. The x-axis is p_L and each series in the graph correspond to different values of p_I.

of the trace generated by Liblog depends on the messages sent and received. The trace size due to messages sent, δ_S is given by Eq. 9.

$$\delta_S = \alpha\, b_F \tag{9}$$

The messages received generate the same size trace entry irrespective of whether the message was received in order or not. The size of the trace generated at a node due to messages received by it is given by Eq. 10.

$$\delta_R = (1 - p_L)\, \alpha\, b_F \tag{10}$$

The size of the trace generated in a checkpoint at a node, δ_N is shown in Eq. 11, where n_P is the number of partners a node communicates with in a checkpoint interval and is shown in Eq. 5.

$$\delta_N = (\delta_S + \delta_R)\, n_P = n_P\, \alpha\, b_F\, (2 - p_L) \tag{11}$$

C. Comparison

Next we compare the trace sizes generated at a single node by CADeT and Liblog to quantify the advantage of CADeT. The key difference between these two approaches stems from the number of bits used to store a trace entry in the common case, namely, b_I for CADeT and b_F for Liblog as we show below.

The ratio of γ_N and δ_N is shown in Eq. 12.

$$\frac{\gamma_N}{\delta_N} = \frac{b_I}{b_F} + \left(\frac{b_A + b_P}{b_F}\right) T_1 + \frac{b_S}{b_F}\, T_2 \tag{12}$$
$$T_1 = \frac{1}{\alpha\,(2 - p_L)}, T_2 = \frac{(1 - p_I)(1 - p_L)}{2 - p_L}$$

The smaller the ratio, the better it is for CADeT. We observe that the ratio is heavily dependent on $\frac{b_I}{b_F}$ because terms T_1 and T_2 are usually small. Since the number of messages per checkpoint interval α is large, T_1 is usually small. To see the effect of p_L and p_I, we varied p_L and plotted the ratio for different values of p_I, which is shown in Figure 3. We used the following values from our empirical evaluation for other variables, namely $b_i = 1$, $b_f = 6$, $b_s = 3$, $b_a = 2$, $b_p = 2$ and $\alpha = 240$. As mentioned above, irrespective of the message losses and out-of-order message arrivals, the size of CADeT traces is very small (15% to 40%) compared to the size of Liblog traces. First consider the scenario of low message losses ($p_L \leq 0.1$). When the messages arrive in order ($p_I \geq 0.9$), CADeT generates smaller size trace because of its efficient representation of common case. CADeT trace size is only about 20% of Liblog trace size. When the messages arrive out of order ($p_I \leq 0.1$), CADeT trace size is about 40% of Liblog trace size. The competitive advantage of CADeT decreased in this case because out-of-order message arrivals require recording additional information such as recording full sequence numbers per out-of-order message arrival. Next, consider the scenario of high message losses ($p_L \geq 0.9$). Irrespective of the order of message arrivals, CADeT trace size is only 15% to 20% of Liblog trace size, which is somewhat counterintuitive. The reason for this is that the lost messages do not generate trace entries at the receiver but do have trace entries at the sender. Since the messages are sent in order, CADeT compresses trace entries corresponding to message sends very well as opposed to Liblog. The few messages that do get delivered at the receiver cause only small increase in trace size for CADeT.

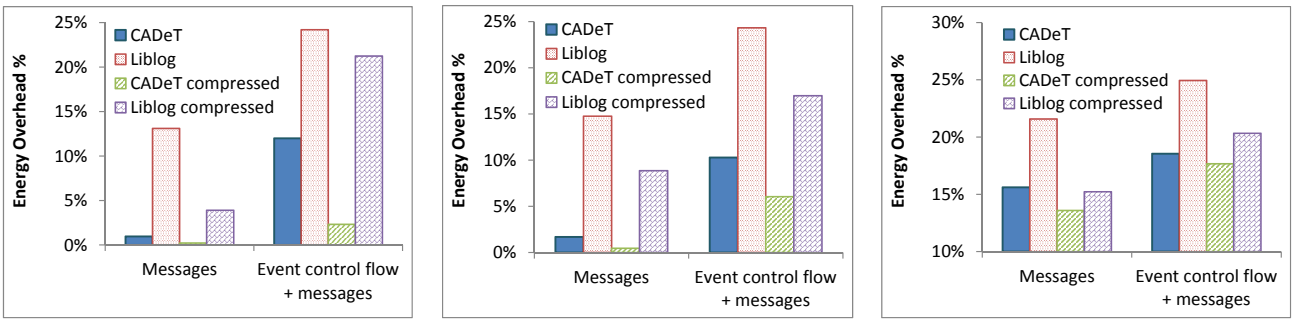
VI. Empirical Evaluation

In order to empirically demonstrate our claims, we implemented our technique CADeT as well as Liblog in TinyOS, a widely used WSN operating system. Results show that CADeT saves considerable energy at a moderate increase in program memory and data memory. We first present the evaluation methodology, followed by energy overhead savings and finally memory overhead.

A. Methodology

1) Metrics: We used the following three metrics to evaluate our approach: (1) energy overhead, (2) program memory, and (3) data memory. Energy overhead corresponds to the additional energy required for tracing and is represented as a percentage of energy consumed by an application without tracing. Lower overhead is better. Observe that the savings in energy overhead are mainly due to trace size reductions and therefore, trace size reduction is implicit in energy overhead savings obtained. Memory is a precious resource for WSNs and both program memory and data memory are very limited. Since saving traces to flash occurs continuously while traces are collected only upon error detection/suspicion, we omit the latter overhead. Furthermore, there are techniques [47] to reduce the trace collection overhead.

2) Benchmarks: For our benchmarks, we used three well-known representative WSN applications which are packaged with TinyOS. First, `Oscilloscope` is a data collection application with high sensing rate (8 times a second). The sensor samples are stored in a buffer and the buffer is sent to the base station when full. Second, `Surge` is another data collection application with medium sensing rate (1 in two seconds). However, `Surge` is a more complex application as it supports sophisticated routing and a query interface to respond to base station queries. Finally, `CntToRfmAndLeds` is a counter application that receives and broadcasts counter values. We use `oscil`, `surge` and `count` respectively to refer to these benchmarks.



(a) Oscil (b) Count (c) Surge

Fig. 4. Energy overhead due to CADeT and Liblog as a percentage of energy consumption without tracing. The x-axis shows two configurations of tracing, one, when just the messages are traced and the other, when control-flow of the send and receive events along with messages are traced.

3) Simulation: We implemented our technique using TinyOS 1.x. While we tested our solution on real test beds, for ease of measurement and parameter tuning, we used TOSSIM simulator which allows simulating the code that runs on motes. TinyOS 1.x was used because we used the open source TinyTracer [48] to generate interprocedural control flow which uses TinyOS 1.x. Our implementation is however version-agnostic and can be used in TinyOS 2.x without any modifications. The energy overheads are measured using PowerTOSSIM and the memory overheads are reported by avr-gcc compiler (TinyOS does not support dynamic allocation).

We ran our simulations for 7.5 minutes and 20 nodes including the base station node. The traces generated are stored in the external flash of the motes. The energy overhead and memory overhead reported are the averages over all nodes. As we noted in Section IV, for several fault diagnostics, the control flow of send and receive events, which give hints on the message contents such as type of the message, is very helpful. Therefore, we show the tracing overhead for two configurations: (a) just messages are traced and (b) both messages and control-flow of send and receive events are traced. By send event control-flow, we mean the control-flow of the `send` and `sendDone` functions at the network layer. By receive event control flow, we mean the control flow of the receive event handler at the network layer and the application layer.

B. Energy Overhead Savings

Figure 4 shows the energy overhead due to CADeT and Liblog when only messages are traced and when messages and send/receive event control flow are traced for all three benchmarks. We used a well-known trace compression algorithm called FCM (finite context method) [49], [50] to compress the traces before writing to the flash. We applied trace compression for both CADeT and Liblog. These results are shown as CADeT compressed and Liblog compressed.

We first observe that CADeT significantly reduces the energy overhead (up to 18× smaller) of message tracing compared to Liblog for all benchmarks. The savings increase when trace compression is used. Since CADeT uses fewer bits to record message sends and receives, CADeT savings accumulate as many messages are exchanged over time. Furthermore, CADeT encodes the entries such that they can be highly compressed as opposed to Liblog, which uses clocks that are not easily compressible.

Figure 5 compares Liblog's and CADeT's energy overhead directly by showing Liblog's energy overhead as a percentage of CADeT's energy overhead. Liblog uses up to 1366% of the energy used by CADeT for the uncompressed case and up to 1918% the energy of CADeT for the compressed case. As explained before, the energy savings are more pronounced in the compressed tracing case because of the compression awareness of CADeT traces. The smaller benefits for `Surge` is due to its low duty cycle, which means `Surge` is mostly idle and sends fewer messages, compared to others.

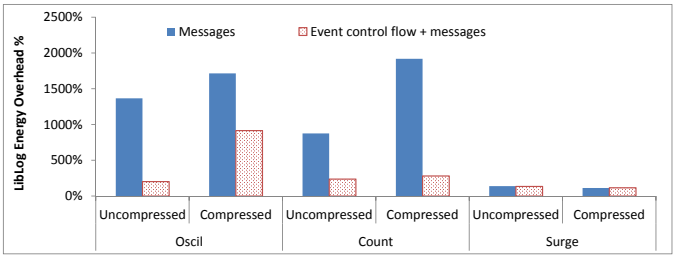
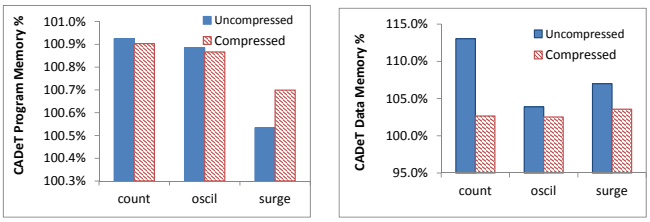


Fig. 5. Comparison of Liblog and CADeT energy overheads. Liblog energy overhead is represented as percentage of CADeT energy overhead.

C. Memory Overhead

Since motes use Harvard architecture, which has separate program memory and data memory (RAM), we measured overheads on program memory and data memory. Data memory is extremely limited (4KB in Mica motes and 10KB in Telos motes). Program memory is limited too (128KB in Mica motes and 48KB in Telos motes).

Figure 6 shows program memory and data memory requirements for CADeT as a percentage of corresponding requirements for Liblog. CADeT uses slightly more program memory (0.3% to 1%) due to it's complex implementation compared to Liblog. This increase is negligible. CADeT uses 3% to 13% more data memory than Liblog. The reason for the increase in data memory is that unlike Liblog, CADeT stores tables such as AAMap and PCMap in the memory. These additional requirements translate up to 300 bytes, which is moderate.



(a) Program memory (b) Data memory

Fig. 6. Memory usage of CADeT as a percentage of memory usage of Liblog

VII. Related Work

We discuss runtime debugging techniques proposed for traditional distributed systems as well as for WSNs. The runtime debugging techniques can be divided into offline techniques and online techniques based on whether the debugging is done postmortem or not.

Online monitoring [10], [51] and predicate detection [11] techniques use external or internal [51] monitoring agents observing the execution of the distributed system, e.g., by snooping messages. Similar solutions for WSNs have been proposed [52], [53]. While these techniques can give insight into the network in small-scale test deployments, they are not cost-effective for large WSN deployments as they require extra hardware – in some cases more powerful than the motes themselves. Furthermore, coordinating the monitors in a network as well as maintaining their correctness is non-trivial in large deployments. Remote debugging tools such as Marionette [19], Clairvoyant [20] and Hermes [21] allow the developer to examine the state of individual nodes and modify the behaviors of the nodes. HSEND [54] is an invariant-based application-level runtime error detection tool for WSNs. The invariants are checked close to the source of the error avoiding periodic collection of data at the base station. An alert is sent to the base station only when there is a violation. Hermes [21] is similar to HSEND [54] but allows developers to modify invariants at runtime as well as deploy patches to fix violations. These approaches are complementary to our diagnostic tracing. Their limitation is that only the faults violating invariants can be diagnosed, and knowledge of failures is required for writing invariants.

Many *offline*, trace-based debugging approaches have been proposed for traditional (wired) distributed systems [13], [14], [15], [55], [30], [16], [18], which use model-based approaches, statistical approaches, or execution replay for diagnosis. Such techniques are inapplicable in the WSN domain due to the extreme resource constraints. In many of the above techniques [16], [18], [13], [30], [14], messages are traced by recording the contents of the messages along with the timestamps generated by Lamport clocks [32]. Such traces can recreate the causal ordering of messages. Netzer et al. [35] observed that only racing messages need to be recorded as others can be regenerated. They presented a mostly optimal tracing technique that uses vector clocks [34] to identify racing messages online and record them. As mentioned before, vector clocks are too heavy-weight for WSNs, and logical clocks assume the presence of a reliable messaging layer such as TCP. Furthermore, logical clocks have high variability, which reduces opportunities for compression.

Sympathy [23] periodically collects WSN information from all nodes at the basestation. The collected information is analyzed to detect node and link failures or partitions and localize their causes. PAD [24] is similar to Sympathy but uses Bayesian analysis to reduce network monitoring traffic.Both Sympathy and PAD require collecting data often, even during correct operation. Moreover, the diagnosis is coarse-grained and is specific to node/link failures but cannot help much with complex faults like data races. NodeMD [26] records system calls and context switches encoded in few bits to detect stack overflows, livelocks, and deadlocks. LIS [27] proposes a log instrumentation specification language and runtime for systematically describing and collecting execution information. LIS is optimized to collect function calls as well as control-flow traces efficiently. TinyTracer [9] proposes an efficient way to record all concurrent events and the interprocedural control-flow paths taken during their execution succinctly. TinyLTS [22] proposes an efficient `printf` logging mechanism that allows the developer to log any runtime information. None of these four approaches handles node *interactions*. While the message sends and receives can be recorded locally, the ordering of messages cannot be recorded. Declarative TracePoints [25] provide a uniform SQL-based query language interface for debugging and can simulate other trace-based approaches. Macrodebugging [8] records traces of all variable values in a macro program of every node in the network. Macrodebugging works at macro level (network level) and cannot be used to diagnose faults at nodes. Unlike [7], CADeT does not require multiple reproductions of faults as well as instrumentation of specific events. We observe, however, that our approach is complementary to the machine learning techniques proposed in [7] as CADeT traces can as well be used in [7].

VIII. Conclusions

In this paper, we proposed a message tracing scheme to record the distributed control-flow that is effective in diagnosis of complex distributed faults in WSNs while satisfying the resource constraints of WSNs. We have shown its effectiveness and argued for its efficiency and consistency advantages over the state of the art.

Acknowledgement

This research is supported, in part, by the National Science Foundation (NSF) under grant 0834529. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and Yield in a Volcano Monitoring Sensor Network," in *USENIX OSDI*, 2006.

[2] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments," in *ACM SenSys*, 2008.

[3] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, "Health Monitoring of Civil Infrastructures using Wireless Sensor Networks," in *ACM/IEEE IPSN*, 2007.

[4] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis, "Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea," in *ACM SenSys*, 2005.

[5] J. Beutel, K. Römer, M. Ringwald, and M. Woehrle, "Deployment Techniques for Wireless Sensor Networks," in *Springer Sensor Networks: Where Theory Meets Practice*, G. Ferrari, Ed., 2009.

[6] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler, "An Analysis of a Large Scale Habitat Monitoring Application," in *ACM SenSys*, 2004.

[7] M. Khan, T. Abdelzaher, and K. Gupta, "Towards Diagnostic Simulation in Sensor Networks," in *IEEE DCOSS*, 2008.

[8] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse, "Macrodebugging: Global Views of Distributed Program Execution," in *ACM SenSys*, 2009.

[9] V. Sundaram, P. Eugster, and X. Zhang, "Efficient Diagnostic Tracing for Wireless Sensor Networks," in *ACM SenSys*, 2010.

[10] K. Sen, A. Vardhan, G. Agha, and G. Rosu, "Efficient Decentralized Monitoring of Safety in Distributed Systems," in *IEEE ICSE*, 2004.

[11] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: Debugging Deployed Distributed Systems," in *USENIX NSDI*, 2009.

[12] H. Garcia-Molina, F. Germano, and W. H. Kohler, "Debugging a Distributed Computing System," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 2, 1984.

[13] A. V. Mirgorodskiy and B. P. Miller, "Diagnosing Distributed Systems with Self-propelled Instrumentation," in *Springer Middleware*, 2008.

[14] N. Maruyama and S. Matsuoka, "Model-Based Fault Localization in Large-Scale Computing Systems," in *IEEE IPDPS*, 2008.

[15] P. C. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior," *ACM Trans. Comput. Syst.*, vol. 13, no. 1, 1995.

[16] R. Curtis and L. D. Wittie, "BUGNET: A Debugging System for Parallel Programming Environments," in *IEEE ICDCS*, 1982.

[17] T. J. LeBlanc, J. Crummey, and M. M., "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computer*, vol. 36, no. 4, 1987.

[18] D. Geels, G. Altekar, P. Maniatis, and T. Roscoe, "Friday: Global Comprehension for Distributed Replay," in *USENIX NSDI*, 2007.

[19] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks," in *ACM/IEEE IPSN*, 2006.

[20] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks," in *ACM SenSys*, 2007.

[21] N. Kothari, K. Nagaraja, V. Raghunathan, F. Sultan, and S. Chakradhar, "Hermes: A Software Architecture for Visibility and Control in Wireless Sensor Network Deployments," in *ACM/IEEE IPSN*, 2008.

[22] R. Sauter, O. Saukh, O. Frietsch, and P. J. Marrón, "TinyLTS: Efficient Network-Wide Logging and Tracing System for TinyOS," in *IEEE INFOCOM*, 2011.

[23] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the Sensor Network Debugger," in *ACM SenSys*, 2005.

[24] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong, "PAD: Passive Diagnosis for Wireless Sensor Networks," in *ACM SenSys*, 2008.

[25] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks." in *ACM SenSys*, 2008.

[26] V. Krunic, E. Trumpler, and R. Han, "NodeMD: Diagnosing Node-Level Faults in Remote Wireless Sensor Systems," in *ACM MobiSys*, 2007.

[27] R. Shea, M. Srivastava, and Y. Cho, "Scoped Identifiers for Efficient Bit Aligned Logging," in *IEEE DATE*, 2010.

[28] M. M. H. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han, "Dustminer: Troubleshooting Interactive Complexity Bugs in Sensor Networks," in *ACM SenSys*, 2008.

[29] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay Debugging for Distributed Applications," in *USENIX ATC*, 2006.

[30] Z. Chen, Q. Gao, W. Zhang, and F. Qin, "FlowChecker: Detecting Bugs in MPI Libraries via Message Flow Checking," in *ACM SC*, 2010.

[31] R. Shea, "Defect Exposure in Wireless Embedded Systems," Ph.D. dissertation, UCLA, 2010.

[32] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Commun. ACM*, vol. 21, no. 7, 1978.

[33] M. Ronsse, K. De Bosschere, M. Christiaens, J. C. de Kergommeaux, and D. Kranzlmüller, "Record/Replay for Nondeterministic Program Executions," *Commun. ACM*, vol. 46, no. 9, 2003.

[34] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *IEEE Parallel and Distributed Algorithms*, 1989.

[35] R. H. B. Netzer and B. P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," in *ACM/IEEE SC*, 1992.

[36] F. J. Torres-Rojas and M. Ahamad, "Plausible Clocks: Constant Size Logical Clocks for Distributed Systems," *Springer Distributed Computing*, vol. 12, no. 4, 1999.

[37] R. Prakash and M. Singhal, "Dependency Sequences and Hierarchical Clocks: Efficient Alternatives to Vector Clocks for Mobile Computing Systems," *Kluwer Wirel. Netw.*, vol. 3, no. 5, 1997.

[38] C. Fidge, "Fundamentals of Distributed System Observation," *IEEE Software*, vol. 13, no. 6, 1996.

[39] T. Ball and J. R. Larus, "Efficient Path Profiling," in *IEEE Micro*, 1996.

[40] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," *IEEE Communications Magazine*, vol. 40, no. 8, 2002.

[41] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless Sensor Network Survey," *Elsevier Computer Networks*, vol. 52, no. 12, 2008.

[42] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan, "An Application-Specific Protocol Architecture for Wireless Microsensor Networks," *IEEE Transactions on Wireless Communications*, vol. 1, no. 4, 2002.

[43] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh, "VigilNet: An Integrated Sensor Network System for Energy-efficient Surveillance," *ACM Trans. Sen. Netw.*, vol. 2, no. 1, 2006.

[44] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y.-R. Choi, T. Herman, S. S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita, "A Line in the Sand: A Wireless Sensor Network for Target Detection, Classification, and Tracking," *Elsevier Computer Networks*, vol. 46, no. 5, 2004.

[45] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler, "Trio: Enabling Sustainable and Scalable Outdoor Wireless Sensor Network Deployments," in *ACM/IEEE IPSN*, 2006.

[46] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva, "Directed Diffusion for Wireless Sensor Networking," *IEEE/ACM Trans. Netw.*, vol. 11, no. 1, 2003.

[47] V. Sundaram, S. Bagchi, Y.-H. Lu, and Z. Li, "SeNDORComm: An Energy-Efficient Priority-Driven Communication Layer for Reliable Wireless Sensor Networks," in *IEEE SRDS*, 2008.

[48] V. Sundaram, "TinyTracer Implementation. http://sss.cs.purdue.edu/projects/TinyTracer/index.html," 2011.

[49] Y. Sazeides and J. E. Smith, "The predictability of data values," in *IEEE Micro*, 1997.

[50] V. Sundaram, P. Eugster, and X. Zhang, "Prius: Generic Hybrid Trace Compression for Wireless Sensor Networks," in *ACM SenSys*, 2012.

[51] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach," in *IEEE SRDS*, 2004.

[52] B. Chen, G. Peterson, G. Mainland, and M. Welsh, "LiveNet: Using Passive Monitoring to Reconstruct Sensor Network Dynamics," in *IEEE DCOSS*, 2008.

[53] M. M. H. Khan, L. Luo, C. Huang, and T. F. Abdelzaher, "SNTS: Sensor Network Troubleshooting Suite," in *IEEE DCOSS*, 2007.

[54] D. Herbert, V. Sundaram, Y. H. Lu, S. Bagchi, and Z. Li, "Adaptive Correctness Monitoring for Wireless Sensor Networks Using Hierarchical Distributed Run-Time Invariant Checking," *ACM Trans. Auton. Adapt. Syst.*, vol. 2, no. 3, 2007.

[55] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: A Pervasive Network Tracing Framework," in *USENIX NSDI*, 2007.