2015 44th International Conference on Parallel Processing

# Connecting the Dots: Reconstructing Network Behavior with Individual and Lossy Logs

Jiliang Wang*, Xiaolong Zheng*, Xufei Mao*, Zhichao Cao*, Daibo Liu† and Yunhao Liu*
* School of Software and TNLIST, Tsinghua University
† School of CSE, University of Electronic Science and Technology of China
{jiliang, xiaolong, xufei, caozc, yunhao, dbliu}@greenorbs.com,

***Abstract*—In distributed networks such as wireless ad hoc networks, local and lossy logs are often available on individual nodes. We propose *REFILL*, which analyzes lossy and unsynchronized logs collected from individual nodes and reconstructs the network behaviors. We design an inference engine based on protocol semantics to abstract states on each node. Further we leverage inherent and implicit event correlations in and between nodes to connect interference engines and analyze logs from different nodes. Based on unsynchronized and incomplete logs, *REFILL* can reconstruct network behavior, recover the network scenario and understand what has happened in the network. We show that the result of *REFILL* can be used to guide protocol design, network management, diagnosis, etc. We implement *REFILL* and apply it to a large-scale wireless sensor network project. *REFILL* provides a detailed per-packet tracing information based on event flows. We show that *REFILL* can reveal and verify fundamental issues, like locating packet loss positions and root causes. Further, we present implications and demonstrate how to leverage *REFILL* to enhance network performance.**

## I. Introduction

### A. Background

In many distributed networks, local and lossy logs are often available on individual nodes. For example, in wireless sensor networks and data center networks, local logs are also commonly available and may also be lossy due to log-write failure or even node failure. On the other hand, those local logs are an important resource, if not the only, to recover the in-situ network status for further network management, diagnosis and other purposes. For example, in wireless ad hoc networks, wireless nodes are relatively isolated individuals connected by fragile and lossy wireless links, making it difficult to learn about the accurate and complete running status of individual nodes in a real-time manner. Currently, there are many real-time network measurement methods and tools proposed in the literature [8] [16] [20] [5] [9] [14]. However, collecting measurement information from the network with data traffic may impact network performance. Thus more information are recorded on local logs and retrieved later. Thus local log is an important resource to understand what has happened in the network.

### B. Motivation

Considering the detailed information recorded in local logs, our main idea is to design and exploit local log files (originally saved at individual nodes, commonly available on different distributed systems [17], containing a collection of events and far from being fully utilized) to recover information from the network. By combining local logs from different nodes, we can further reconstruct the network-wide behavior that is otherwise difficult to reveal. We define the network-wide behavior as network *event flows*, i.e., sequences of events according to their occurrence in the network. The event flow reflects the running status of the network. It can also provide information for network diagnosis and answers to questions such as "what's happened on earth in the network?".

To reconstruct network behavior, we need to exploit local logs and combine logs from different nodes. Though there are a large collection of approaches for log analysis, existing approaches usually focus on analyzing local logs from a single node (e.g., PC), or assume a complete and correct log [21] [17] [19] [18]. Those approaches cannot be used to reveal information with lossy logs from different nodes in distributed networks. Wit [10] proposes a method to analyze the sniff logs from different nodes in the network. However, Wit exploits sniff logs collected by multiple overhearing nodes. Different overhearing nodes may receive the same packets and thus common events are recorded. Wit can use those common recorded events to synchronize and combine logs from different nodes. In other scenarios, logs recording local events on individual nodes contain no such kind of commons events and thus logs cannot be combined.

To analyze lossy logs from different nodes, there are several challenges. First, each individual node only has a local view of its state. Logs collected from different nodes are usually not synchronized. Logs should be combined to derive a complete view of the entire network. Second, unlike logs assumed to be recorded on a reliable node and collected through reliable links, logs in distributed networks may be incomplete due to reasons such as malfunction of nodes (as commonly observed in distributed system), fragile links, wireless interference, buffer limitation and so on. Third, logs should be analyzed according to protocol semantics. Protocol interactions among different nodes should be considered while analyzing logs from different nodes.

Our approach exploits the implicit and inherent network behavior correlations. We find that implicit network correlations, which provide useful information to combine different logs, can also be used to recover lossy logs and combine

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.26

logs from different nodes. First, there are intra-node behavior correlations for operations on a single node. For example, a sending operation on a node may imply that the corresponding packet is already received and the receiving operation is already performed. Those correlations can be leveraged to reveal the event flow and infer lost events (e.g., the receiving event) on a single node. Second, logs from multiple nodes are also implicitly correlated according to inter-node event correlations. For example, a receiving operation on a receiver may imply a corresponding sending operation on the corresponding sender. Those correlations can be used to infer event ordering between nodes and related lost events. Such intra-node and inter-node correlations, while containing important information, are not fully exploited in existing approaches. We leverage those intrinsic correlations to recover the event flows.

This work is also motivated by a real project *CitySee* [11], a $CO_2$-monitoring wireless sensor network in an urban area consisting of 1200 sensor nodes. While maintaining the network, we find that the data delivery performance fluctuates and there are many packet losses in the network. However, based on collected data packets (including both sensory data and status data), we can only obtain the macroscopic view of the data, leading to challenges to diagnose and improve the network. For example, although we know a portion of packets are lost, it is difficult to know why those packets are lost. On the other hand, we record events on individual nodes as local logs. Those local logs are not fully explored. We resort to leverage those event logs on individual nodes to provide information to answer questions for network diagnosis.

### C. Our Approach

In this work, we propose *REFILL*, an event flow based approach contributing to fine-grained network management such as network diagnosis and network measurement through exploiting local logs on individual nodes. *REFILL* first derives an inference engine based on finite state machine to model single node operations. Then *REFILL* connects inference engines on different nodes based on implicit network event correlations. Therefore, *REFILL* synchronizes logs from different nodes with correlated events, and infers lost events for incomplete logs by leveraging implicit intra-node and inter-node event correlations. Then *REFILL* reconstructs event flows in the network (e.g., packet tracing can be depicted as a series of event flows), and derives network diagnosis information. Unlike traditional log analysis approaches which usually assume a complete and synchronized log from a single node, *REFILL* processes lossy and unsynchronized logs collected from different nodes in a distributed environment and provides network related information that are otherwise difficult to obtain.

The contribution of the paper is summarized as follows.

- *Event analysis and event flow recovery with individual lossy and unsynchronized logs.* We propose to use local logs for revealing event flows in large-scale distributed networks. To work with unsynchronized and incomplete logs from different nodes, we design an inference engine based on finite state machine to abstract states on each node. We formally define inherent and implicit event correlations in and between nodes, and leverage event correlations to synchronize different nodes and infer lost events.
- *Implementation of REFILL system.* We implement *REFILL* to evaluate the performance in wireless sensor networks. The implementation consists a component on sensor node and a component on the PC for analysis.
- *Application to a real distributed network.* We apply *REFILL* to CitySee, a large scale wireless sensor network. We show that *REFILL* can provide fine-grained network management (e.g., efficient packet tracing and locating causes of packet losses). Further, we present the implications of *REFILL* to wireless sensor network design.

The organization of the remaining part is as follows. Section II shows the network model and the problem definition. Section III introduces the challenges for *REFILL* design. Section IV introduces the *REFILL* system design. Section V presents the implementation and evaluation results of *REFILL*. Section VI presents the related work. Section VII concludes this work.

TABLE I: Examples of events in the network

| Event | Description |
|---|---|
| $n_1 - n_2$ recv | The packet from $n_1$ is received at node $n_2$. Recorded on node $n_2$. |
| $n_1 - n_2$ overflow | There is no space on node $n_2$ for the packet from $n_1$ and thus the packet has to be discarded. Recorded on node $n_2$. |
| $n_1 - n_2$ dup | A duplicated packet is received by $n_2$ from $n_1$. Each node keeps a buffer of received packets and duplication event happens when a received packet is identical to any packet in the buffer. The duplication event is often due to routing loops. |
| $n_1 - n_2$ trans | The packet is transmitted by $n_1$ to $n_2$. Recorded on node $n_1$. |
| $n_1 - n_2$ ack recvd | the packet from $n_1$ to $n_2$ is acked by the receiver, i.e., an acknowledgement is received. Recorded on node $n_1$. |

## II. Network Model and The Problem

Logs are commonly recorded on individual nodes for network management, diagnosis, etc. Usually, a log statement in the program records a corresponding event. For example, in data transmission scenario, a log statement may be added to the position where a packet is sent and thus the sending operation is logged as an event. More specifically, an event indicates that the program has reached the corresponding position in the program and it has corresponding related information upon reaching the position. Intuitively, we can see that the ordering of event occurrence on different nodes can be used to recover the program running status and network behavior. The related information may contain different kinds of information depending on the log operations. For example, for a sending event, the sender and receiver may be recorded as the related information. When an error is recorded as an event, the corresponding error type may be recorded as the

related information. In our approach, we do not have specifical requirement for the related information in the log. We use the related information to find the correlation between nodes. Given more related information, more correlations between nodes can be found. However, it is not mandatorily required for each event.

We denote an event as a tuple $E = (V, L, I)$, where $V$ denotes the event type, $L$ denotes the location where the event happens and $I$ is the related information, e.g., the sender and receiver information for the sending event. Such kind of event is often available in practical system logs [21] [17] [19] [18]. $I$ can also be empty for some event types. The event occurrence time is not required to be logged. This is practical for real systems, e.g., different nodes may not be precisely synchronized in distributed system. For simplicity of presentation, we implicitly add the occurrence time $T$ for each event. We assume a network consists of $N$ nodes, denoted as $\{1, 2, \ldots, N\}$. Each node in the network can log events locally. We use $E_{i,j} = (V_{i,j}, L_{i,j}, I_{i,j}, T_{i,j})$ to denote the $j$th event on node $i$. An event flow is an ordering of all events. Denote $\tilde{F}$ as the event flow, we should have

$$\tilde{F} = E_{i_1,j_1}, E_{i_2,j_2}, \ldots, E_{i_K,j_K} \quad (1)$$

where $\forall 1 \leq m < n \leq K$, we have $T_{i_m,j_m} < T_{i_n,j_n}$.

For example, a packet in the network can trigger different events, such as sending event, receiving event and etc. The event flow is to recover the correct order of all the events related to the same packet in the network. With the event flow, the detailed behavior of the packet can be revealed, e.g., the path of the packet, where the packet is lost and the occurrence of loop for the packet can be revealed. Further, the packet related information, e.g. per-packet delay, packet retransmission, packet loss, can also be revealed. The protocol behavior for different nodes in the network can be recovered with the event flow.

In practical networks, according to different event types, there may be different related information for different events. For example, a network event has information about related packets and related nodes. Many network operations have two related nodes, i.e., the sender and the receiver, denoted as sender−receiver. For example, Table I shows some examples of events for packet reception, queue overflow, packet duplication, packet transmission, and packet acknowledgement. Since the sender or receiver already contains the information of $L$, hereafter, for brevity, we use (sender−receiver, event type, $L$) to denote an event. $L$ is omitted when there is no ambiguity.

## III. Challenges

In practical networks, there are several challenges to analyze event logs. We show with examples for the challenges.

**Incomplete information.** First, the logs collected from different nodes are incomplete. Intuitively, a straightforward method to analyze the logged events is based on the protocol semantics. For example, if a node $n$ records a trans event and does not have an ack event for a packet. This packet is considered lost on node $n$ since the acknowledgement (ack) of the packet is not received. However, this may not always be true. The logs may be incomplete and the ack event may be lost. We need to correctly analyze the events even with incomplete logs.

TABLE II: A simple example with logs from 3 nodes.

| Case | Node 1 | Node 2 | Node 3 |
|---|---|---|---|
| complete log | 1−2 trans<br>1−2 ack recvd | 1−2 recv<br>2−3 trans<br>2−3 ack recvd | 2−3 recv |
| Case 1 | 1−2 trans | Lost | 2−3 recv |
| Case 2 | 1−2 trans<br>1−2 ack recvd | | |
| Case 3 | 1−2 ack recvd<br>1−2 trans | | |
| Case 4 | 1−2 trans<br>1−2 ack recvd<br>3−1 recv<br>1−2 trans<br>1−2 ack recvd | 1−2 recv<br>2−3 trans<br>2−3 ack recvd<br>2−3 trans | 2−3 recv<br>3−1 trans<br>3−1 ack recvd |

As in Table II, we list examples of events collected from different nodes. The first row shows the case for a complete log of events on three nodes (node 1, 2 and 3). For case 1, events from node 2 are lost and only events from node 1 and 3 are collected. It is difficult to derive the event flow with traditional approaches and derive further information. For example, the packet can be considered lost on node 1 since there is no ack event. In fact, this packet is not lost on node 1 since node 3 has received the packet from node 2. The key insight here is that an event can be used to imply other useful information. A recv event on node 3 implies that the corresponding packet is received on node 2.

**Unsynchronized events.** The collected events are not synchronized, resulting in different event ordering in event flow. The order of events on each node determines the real network behavior. For example, it is normal that a trans event is followed by an ack event. As shown in case 2, an ack event is followed by a trans event. This is significantly different from case 3. In case 3, an ack event precedes the trans event. In such a case, node 1 should receive and forward the packet twice, which actually indicates duplications or routing loops in the network. When there are unsynchronized information from different nodes, it is difficult to analyze the events.

**Distributed environment.** Moreover, events are logged and collected from different nodes in a distributed manner. Information from different nodes should be connected and combined to derive a correct network-wide event flow. Even when there is no lost event, it is still difficult to analyze events from different nodes. As in case 4, it seems that all transmissions are acked if we solely look at the trans and ack event. In fact, there is a loop in the network if we look at those events and their corresponding occurrence ordering. The packet is lost at node 2 since the second transmission from node 2 to node 3 fails. While processing events from different nodes collected in distributed environment, we should consider the state correlation on different nodes. Otherwise, it is difficult to reveal correct network-wide information.

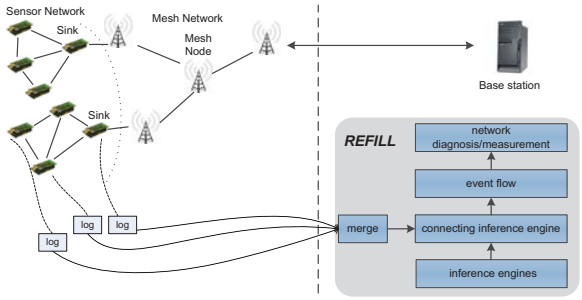


Fig. 1: The *REFILL* system overview.

## IV. System Design

To address those problems, we propose the design of *REFILL*. The system overview of *REFILL* is shown in Figure 1. The first step is to collect events from the network. Logs containing events from different nodes are first merged with ordering of events from the same node preserved. The second step is to associate events with node state. In this step, we build inference engines for different nodes by leveraging intra-node correlations. By intra-node event correlation, we can improve those inference engines and enable them to process lossy logs on individual nodes. The third step is to connect events from different nodes. Based on inherent inter-node correlations, we can connect inference engines from different nodes. The connected inference engine takes the merged events as input and outputs the event flow.

### A. Inference Engine

The first step is to build an inference engine for each node. We use the finite state machine (FSM) based inference engine to model the states on each node. Since the event log is coupled with log statements and their positions that produce the log, the FSM can be generated according to the log positions. The FSM can be generated manually [21] or with automatic tools [6]. The original FSM is generated according to the original program. When there is no lost event, the generated FSM can transit correctly with the collected events. Figure 2 shows a simplified example of two FSMs for inference engine. The FSM is modeled as a directed graph $\mathcal{G} = (\mathcal{S}, \mathcal{T}, \mathcal{E})$, where $\mathcal{S} = s_1, s_2, \ldots, s_n$ are $n$ vertices corresponding to $n$ states, $\mathcal{T}$ are the directed edges and $\mathcal{E}$ are the corresponding events on the edges. In the transition graph, we have

- transition $t_{i,j}$: the edge from $s_i$ to $s_j$, which corresponds to the transition from $s_i$ to $s_j$. A transition $t_{i,j}$ can also be denoted as $s_i \rightarrow s_j$.
- event $e_{i,j}$: the event on edge $s_i \rightarrow s_j$. It should be noted that $e_{i_1,j_1}$ may be equal to $e_{i_2,j_2}$ for $i_1 \neq i_2$ or $j_1 \neq j_2$. This means that multiple transitions may have the same type of event. An event may lead to different transitions.
- transition sequence $TS_n = t_{i_1,i_2}, t_{i_2,i_3}, \ldots, t_{i_{n-1},i_n}$. A transition sequence is a directed path on the transition graph.

We say a state $s_{i_n}$ is reachable from state $s_{i_1}$ (denoted as $s_{i_1} \succ s_{i_n}$) if and only if there is a transition sequence $TS_n$ from $s_{i_1}$ to $s_{i_n}$. This also means there exists a path from state $s_{i_1}$ to $s_{i_n}$ on the transition graph. We denote those transitions on the original FSM as *normal transitions*.

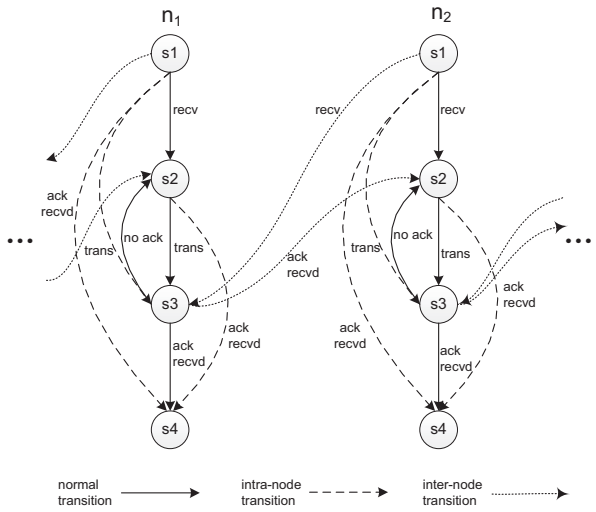


Fig. 2: A simplified example of FSMs for the inference engine. The solid line shows the original FSMs for the inference engine.

More specifically, for a transition graph, we have the following definition.

*Definition 4.1 (prerequisite transition):* we say an event $t_{i_1,j_1}$ is a prerequisite event of $t_{i_2,j_2}$ (denoted as $t_{i_1,j_1} \vdash t_{i_2,j_2}$) if and only if $t_{i_2,j_2}$ can occur only after $t_{i_1,j_1}$ occurs.

Intuitively, if $t_{i_1,j_1}$ is a prerequisite transition of $t_{i_2,j_2}$, it means when $t_{i_2,j_2}$ occurs, $t_{i_1,j_1}$ must have occurred before $t_{i_2,j_2}$. In other words, $t_{i_2,j_2}$ should not occur if $t_{i_1,j_1}$ has not occurred. Meanwhile, the events corresponding to the prerequisite transitions are denoted as prerequisite events. It can also be noted that the prerequisite transition for a particular transition can be on the same node or on another node. We use the prerequisite transitions to build inter-node transitions. It can also be seen that the prerequisite events are due to inherent event correlations.

### B. Transitions in Inference Engines

To combine logs from different nodes, we leverage the network inherent correlations to connect states on the inference engines. Besides the normal transition, there are two types of event connections that we leverage to connect intra-node and inter-node states on inference engines.

**Intra-node transition.** In the original FSM, the state can only correctly transit when there are no lost events. We find that a state on one node may indicate some prior states in the same node. For example, as we have shown, a sending operation on one node actually indicates prior receiving operation for the same packet on the same node.

Given an event $e_{i,j}$, for all transitions $s_{i_1} \rightarrow s_{j_1}, s_{i_2} \rightarrow s_{j_2}, \ldots, s_{i_m} \rightarrow s_{j_m}$ with the same event $e_{i,j}$ and for any state $s_x$ in graph $\mathcal{G}$, if there is one and only one state $s_{j_c}$ in all states

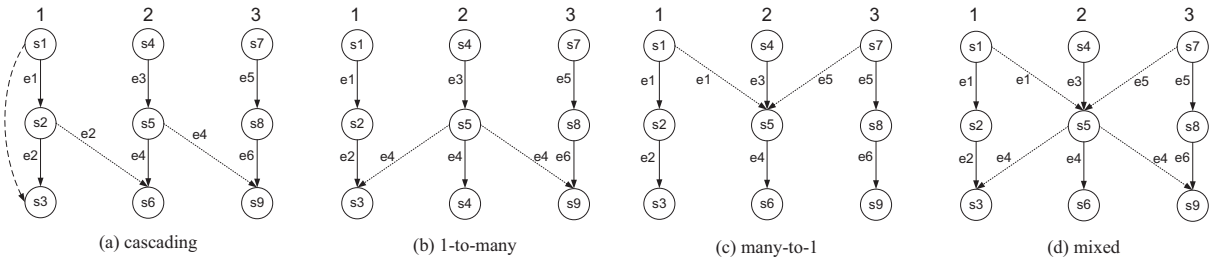


Fig. 3: Event processing according to the transition algorithm and connected inference engine. Figure (a) shows inference engines with cascading inter-node transitions. The resulted event flow is e1, e3, e5, e6, e4, e2. The transition algorithm recursively processes the prerequisite transition to s6 and then s9. Figure (b) shows the inference engines with 1-to-many inter-node transitions. The events $e2$ and $e6$ should occur before $e4$. The ordering between e1 and e5 cannot be determined in this example. Figure (c) shows the inference engines with many-to-1 inter-node transitions. The event $e3$ must occur after $e1$ and $e5$. Figure (d) shows the mixed inter-node transitions.

$s_{j_1}, s_{j_2}, \ldots, s_{j_m}$ that is reachable from $s_x$, i.e., $s_x \succ s_{j_c}$, we can add an intra-node transition from $s_x$ to $s_{j_c}$ with event $e_{i,j}$. Intuitively, this means that for an event $e_{i,j}$ at state $s_x$, this event can only possibly be generated at the transition $s_{i_c} \rightarrow s_{j_c}$ since $s_{j_c}$ is the only reachable state from $s_x$ which has a transition with event $e_{i,j}$. Therefore, if at state $s_x$ an event $e_{i,j}$ occurs, even when there is not available normal transition for event $e_{i,j}$, this implies the state actually reaches $s_{j_c}$. This may happen when there are event losses in the collected logs. Thus we can directly jump from the state $s_x$ to the state $s_{j_c}$ with event $e_{i,j}$. Meanwhile, an intra-node transition from $s_x$ to $s_{j_c}$ also indicates prerequisite events of $e_{i,j}$ on the transition sequence $s_i \succ s_{j_c}$ following normal transitions are lost. In such a case, even in the presence of event losses, the state transition can still move forward.

As shown in Figure 2, there are two simplified FSMs on two nodes. The FSM of node $n_1$ corresponds to the state transition on node 1. The transitions represented with solid line are the normal transitions. The dashed line represented the intra-node transitions we add in the original FSM.

**Inter-node transition.** In order to process events for multiple nodes, we need to connect states on different nodes. For network operations that related to multiple nodes, network operations may change states on different nodes. For example, an event ($n_1 \rightarrow n_2$, recv) indicates that $n_2$ has successfully received this packet. Meanwhile, it also implies $n_1$ has sent the packet. At this time, the state on node $n_1$ can transit to the corresponding state. For an event at node $n_2$, both states on node $n_1$ and $n_2$ should accordingly change. Thus network operations can be used to synchronize states on different nodes.

More specifically, we connect different FSMs based on the prerequisite transitions. For an FSM $F_1$ and a transition $t_{i_1,j_1}$ (i.e. $s_{i_1} \rightarrow s_{j_1}$) with event $e_{i_1,j_1}$ on $F_1$, if transition $t_{i_2,j_2}$ is a prerequisite transition of $t_{i_1,j_1}$ in $F_2$, we can add a transition $s_{i_1} \rightarrow s_{j_2}$ with event $e_{i_1,j_1}$ from $F_1$ to $F_2$. We denote the state $s_{j_2}$ as the *prerequisite state* of $s_{i_1}$. It can be seen that prerequisite transitions should be finished (i.e., prerequisite state is reached) in $F_2$ before moving to the current state $s_{j_1}$ with event $e_{i_1,j_1}$ in $F_1$. It should be noted a state may have both intra-node transition and inter-node transition for the same event. Meanwhile, a state may have inter-node transitions to different nodes with the same event. For example, a broadcast event can lead to state change on different nodes.

Figure 2 also illustrates some inter-node transitions to connect different FSMs. For example, there is an inter-node transition from $s_1$ on $n_2$ to $s_3$ on $n_1$. This is because a recv event on $n_2$ indicates a send event on $n_1$. Otherwise, there should be no recv event on $n_2$. Meanwhile, for event with intra-node transition, there may also exist corresponding inter-node transition. In Figure 2, we omit some of those inter-node transition for brevity.

**Processing Events.** The connected FSMs take events as input, transit on the FSMs and then output the event flow. We process events on the connected FSMs based on normal transition, intra-node transition and inter-node transition. Assume the current processing node is $curNode$, the main steps for the transition algorithm start from a given node and process events recursively as follows.

1) If there is a normal state transition for $curNode$ with the current event $curEvent$, process the corresponding normal state transition and add $curEvent$ to the event flow. Meanwhile, if there is inter-node transition to a state $s_x$ on another node $i$ for the event, recursively process events on the node $i$ until reaching state $s_x$.
2) Otherwise, if there is an intra-node transition, process the intra-node state transition to state $s_x$. If there are prerequisite events for $curEvent$ on $curNode$, we need to add those prerequisite events to the event flow and recursively process those prerequisite events as in step 1). This is because current event $curEvent$ only occurs after all prerequisite events have occurred. Then, we add event $curEvent$ to the event flow. Those prerequisite events corresponds to those lost events and are inferred by our method.
3) If there is no event for $curNode$, we switch to other nodes with unprocessed events. For events that cannot

be processed (i.e., no available transition) for $curNode$, we omit those events on $curNode$.

4) Otherwise, the transition stops.

**Transition Example.** With inter-node transition, events from different nodes are connected. Meanwhile, with prerequisite events, lost events can be inferred from the transitions. In Figure 3, we illustrate how to recursively process events with inter-node transitions.

Figure 3 (a) illustrates inference engines with cascading inter-node transitions. The transition algorithm first processes the event $e1$. While processing $e2$, there is a prerequisite state $s_4$ that should be processed first. Thus the algorithm needs to process $e3$ and $e4$. While processing $e4$, the prerequisite state $s_6$ needs to be processed first. Therefore, event $e5$ and $e6$ are processed. The final obtained event flow is $e1, e3, e5, e6, e4, e2$. Such a case may happen for a multiple hop data transmission in distributed networks, e.g., wireless sensor networks. From this example, we can also see that even when there is only one event $e2$ on node 1 and all other events are lost, the transition algorithm can generate the correct event flow and infer lost events. More specifically, $e1$ can be inferred since it is a prerequisite event for intra-node transition from $s_1$ to $s_3$ with event $e2$. Events $e3, e4, e5, e6$ are inferred with cascaded prerequisite events according to inter-node transition.

Figure 3 (b) illustrates inference engines with 1-to-many inter-node transitions. The events and transitions are the same with those in Figure 3 (a) except in this example there are multiple inter-node transitions for a particular event. In this case, events $e1, e2$ and $e5, e6$ are prerequisite events for $e4$, and should be processed first. Therefore, the resulted event flow can be $e1, e2, e5, e6, e3, e4$. The relative ordering for events between node 1 and node 3 are not determined. This can be used to model the process of node 2 waiting for response from node 1 and 3. For example, this can be a case of data dissemination, in which node 2 are waiting to check whether node 1 and node 3 have received data from node 2.

Figure 3 (c) illustrates inference engines with many-to-1 inter-node transitions. Event $e3$ is the prerequisite event for events $e1$ and $e5$. Thus $e3$ should occur before $e1, e2$ and $e5, e6$.

Figure 3 (d) illustrates inference engines with mixed inter-node transitions. Accordingly, the event flow should satisfy the constraints with mixed inter-node transitions. For example, the inter-node transitions with events $e1$ and $e5$ indicate that $e3$ should occur before $e1$ and $e5$. Inter-node transition $e4$ indicates event $e2$ and $e6$ should occur before $e4$. Such a case can be used to model the negotiation process in which node 2 broadcasts information and then waits for responses from node 1 and node 3.

### C. Event Flow

According to the aforementioned event processing method, we can obtain the event flow in the presence of event loss. We can also infer lost events that are not recorded in the log. Normally, if a transition follows a normal state transition, there is no lost event. For intra-node transition from state $s_x$ to state $s_y$, there may be some lost events. Those prerequisite events can be inferred as lost events. It should be noted that not all lost events can be inferred by the inference engine. Meanwhile, we also show that the resulted event flow and inferred lost events can be used to derive network information and be used for network diagnosis. For example, the results for different cases in Table II are as follows.

- Case 1. The input events are 1−2 trans and 2−3 recv. The output event flow is 1−2 trans, [1−2 recv], [2−3 trans] and 2−3 recv. The events in square brackets [ ] are those lost events inferred with the transition algorithm. The algorithm first processes event 1−2 trans. Then the algorithm processes event 2−3 recv. For event 2−3 recv, event 1−2 recv, and 2−3 send are prerequisite events and thus they are added to the event flow. In this example, not only the lost events [1−2 recv] and [2−3 trans] are inferred, the correct ordering for events are also recovered.
- Case 2. The output event flow is 1−2 trans, [1−2 recv] 1−2 ack recvd. The packet is lost after the packet is successfully transmitted to node 2 since a lost event [1−2 recv] is inferred with the transition algorithm.
- Case 3. The output event flow is [1−2 trans], [1−2, recv], 1−2 ack, 1−2 trans. The lost events [1−2 trans], [1−2, recv] are inferred with event 1−2 ack. The packet is retransmitted from the inferred lost events. Meanwhile, the packet is lost when the packet is transmitting from node 1 to node 2. Unlike traditional analysis, even though there is a pair of trans event and ack recvd event, this does not mean the packet is received at the receiver. The ordering for the trans and ack event impacts the actual event flow as well as the diagnosis result. Based on the event flow, we can investigate the system performance and examine the protocol and program states according to the position where the transition stops in the system.
- Case 4. The output event flow is 1−2 trans, 1−2 recv, 1−2 ack recvd, 2−3 trans, 2−3 recv, 2−3 ack recvd, 3−1 trans, 3−1 recv, 3−1 ack recvd, 1−2 trans, [1−2 recv], 1−2 ack recvd, 2−3 trans. From the lost event 1−2 recv that is inferred with inference engine, we know that the packet is received on node 2. Further, from event 2−3 trans, we can infer the packet is lost when node 2 is transmitting to node 3.

## V. Apply REFILL to a Real Network

We implement *REFILL* on both wireless sensor networks and PC. In WSNs, we implement the event system with NesC language. The event system has no special requirement for event collection. We use the widely adopted data collection protocol CTP [4] to collect events. It should be noted that we do not require to collect complete logs from all nodes. We implement the inference engine and the transition algorithm with Perl language.

We evaluate the performance of *REFILL* from the following aspects:

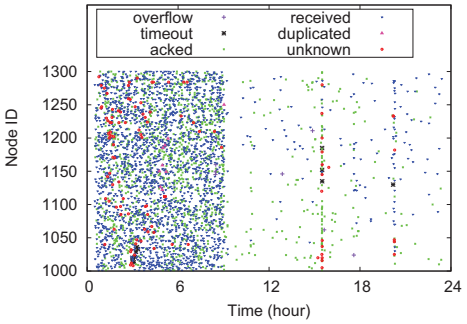


Fig. 4: Sink view of lost packets.

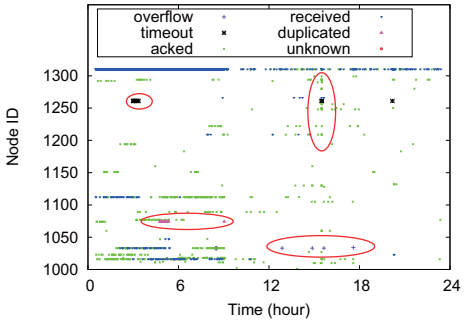


Fig. 5: Causes for lost packets with *REFILL*.

- We evaluate the performance of using *REFILL* for state transition on different nodes to derive the event distribution for lost packets in a real wireless sensor network;
- We further show the performance of using *REFILL* to find the causes for packet losses in a real network.
- We show the implications to network design and measurement based on the results of *REFILL*. We show how to improve the network performance with results from *REFILL*.

### A. Network Protocols

We apply *REFILL* to an outdoor wireless sensor network project, CitySee, which consists of 1200 nodes deployed in an urban area. Figure 8 shows the spatial distribution of sensor nodes. Before introducing the details of the evaluation results, we first briefly introduce the network protocols in the network.

*1) PHY Layer:* At the physical layer, we use the CC2420 radio chip which is compliant with 802.15.4 protocol. The packet contains a PHY header (i.e., the length field of the packet), the payload and a Cyclic redundancy check (CRC). At the receiver, each node first demodulates the PHY header and then receives the packet according to the length. When a packet is entirely received, the PHY layer first checks the CRC. If the CRC check is not passed, the PHY layer simply discards this message. Otherwise, the PHY layer sends an acknowledgement to the sender (in hardware ACK model) and then delivers the packet to the upper layer.

*2) MAC Layer:* The MAC layer header follows the PHY layer header. It mainly contains the length field, sender and receiver ID, the frame check sequence (FCS) and the data sequence (DSN) field. Sensor nodes are battery powered and thus the energy budget is very limited. We use the Low Power Listening (LPL) MAC layer protocol to save energy. The basic mechanism of LPL is as follows. Each node periodically turns on the radio to sense the channel. If the channel is idle, which implies no packet transmission, the node will turn off the radio. If the channel is busy, the node will keep the radio on and then decode the signal to see if it is the intended receiver. If yes, it first sends an acknowledgement and then keeps the radio on for another short period of time for possible consecutive packets. Otherwise, it will turn off the radio. If a node has packets to send, it repeatedly sends the packets until an ACK is received or a timeout of a certain period.

*3) Data Collection:* The data collection protocol is based on the Collection Tree Protocol (CTP) [4]. In CTP, data are collected through a routing tree, which is built based on the *ETX* metric [1]. More specifically, the link ETX is calculated as $1/q$, where $q$ is the link quality. A path ETX is the sum of link ETXs along the path. Each node selects the path with smallest ETX as the routing path and accordingly chooses the parent node. In the network, each node first measures the link ETX to all neighbors. The link ETX between $n_1$ and $n_2$, i.e., linkETX($n_1$, $n_2$), can be calculated by measuring the link quality between $n_1$ and $n_2$. Each node $n$ is initialized with an infinite path ETX value, i.e., pathETX($n$) = $\infty$ except the sink node with a path ETX of 0. To calculate the path ETX value to the sink node, each node broadcasts a routing packet with the path ETX value. Upon receiving a routing packet from node $n_1$, $n_2$ updates the path ETX value to pathETX($n_1$) + linkETX($n_1$, $n_2$) and change the parent to $n_2$ if and only if pathETX($n_2$) $>$ pathETX($n_1$) + linkETX($n_1$, $n_2$).

### B. Network Diagnosis with Event Flow

During the operation of the network, we find there exist a portion of packet losses. To further examine the lost packets to improve the network performance, we apply *REFILL* to the collected log data from the network. We can obtain the event flow for each packet. Further, from the event flows, we can obtain the information where packets are lost and why they are lost. We show the information obtained by applying *REFILL*, the spatial and temporal distribution of packet losses. We show the differences for the spatial and temporal distributions between *REFILL* and other approaches. Further, based on *REFILL*, we examine the causes for packet losses in the network. We also show the implications of the results and how to improve the system performance based on the results.

*1) Temporal Distribution:* We first examine the temporal distribution of lost packets. From the event flow, we can also derive the causes of packet losses from the event flow. For example, we say the cause is received loss if the last event of the packet's event flow is a received event.

Figure 4 shows the temporal distribution of lost packets in the source node's view. This is obtained from the collected data packets by analyzing whose packets are lost. It should be

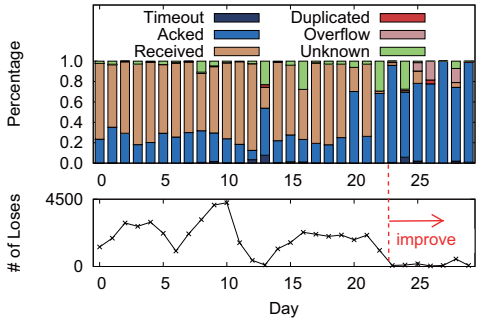


Fig. 6: Percentage of different causes over a month.

Fig. 7: Connection to the pins.

noted that we do not have the received time for lost packets, since those packets are not received at the base station. Thus, we calculate the approximate lost time for those packets as follows. We calculate the time for the received packet right before the lost packet. Then we calculate the sequence gap between the lost packet and the packet before the lost packet. Since packets are sent periodically in our network, we can derive the sent time of lost packets and use it to approximate the packet loss time. The $x$-axis is the time over two days and $y$-axis is the node ID. Different markers represent different causes. We can see that packets generated at different nodes have a similar probability to get lost. Meanwhile, packet losses are strongly temporal correlated. Packet losses often occur at the same time period. We can see that the number of timeout losses and duplicated losses is not high. There are few overflow losses in the network. This is because that the network is not under a high traffic pressure. On the other hand, those timeout losses, though impact different nodes, almost occur at the same time. Meanwhile, there are only few duplicated losses in the network and they also happen at almost the same time period.

However, from Figure 4, we still do not know where packets are lost. We further investigate packet losses in the network with *REFILL*. Figure 5 shows the causes of packet loss according to loss occurrence positions. The loss position is the sensor node where the packet gets lost. The loss causes and loss positions are derived from the event flow of *REFILL*. We find that though the sources of lost packets are evenly distributed, the loss positions are on a small portion of nodes rather than evenly distributed in the entire network. We can also see that timeout and duplicated losses are bursty as shown in those ellipses in Figure 5. We investigate the network and find that this may be due to temporary low quality links or routing loops. Meanwhile, we find that there are a lot of received losses on the sink node (upmost band in the Figure 5).

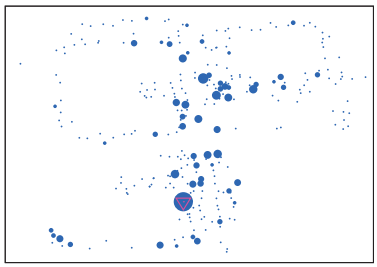

Fig. 8: Spatial distribution of received losses. The radius of circle indicates the number of packet losses. The triangle denotes the sink node.

This indicates that many packets are lost even after they have arrived at the sink node.

We also examine the packet loss causes over 30 days based on the results from *REFILL*. Figure 6 shows the cause composition. For different days, the causes are different and the composition of different causes are also different. First, *REFILL* finds the causes for most lost packets over 30 days. The two most common causes are the acked and received losses. This means that the number of packet losses due to low link quality is not high. We investigate the data and find that those received and acked loss are mostly due to packets loss on the sink. Thus there are many received losses on the sink node as shown in Figure 6. On the 9th and 10th day, the packet losses become high due to snow.

Based on the result, we investigate packet losses on the sink and the received losses and acked losses. We find that this is simply due to the unstable connection from the sink node to the base station. As shown in Figure 7, the sink node is connected to a high speed backbone node (mesh node) using RS232 connection. We directly connect RS232 pins to the chip pins on board. By using the signal from the chip, data are transmitted through the RS232 wire to the mesh node. Though we have conducted extensive experiments in testbed environment, when deployed outside we use a long cable and the signal becomes unstable over the long cable. Hence, many packets from sensor node to base station get lost. This problem, which seems to be subtle, is not revealed until we thoroughly analyze the packet loss position by using *REFILL* on the log data. Meanwhile, it is difficult to reveal the problem since the sink node is installed at a high position that we cannot easily to reach. After the 23th day, we changed the sink and its connection to the mesh node. We can see packet losses are significantly reduced.

*2) Spatial Distribution:* With REFILL, we can also derive the spatial distributions for different types of packet losses. Figure 8 shows the received packet losses, i.e., packet losses even when they are received on a certain node. For example, a packet gets lost while being processed inside a node. It can also be seen that the sink node has a large number of received losses, in which packets get lost even after they have arrived at the sink node. One of the main causes is the sink node design as we have explained. We have fixed this problem after we investigate the sink node.

### C. Cause Inference

We further look into the breakdown of packet losses. Over the 30 days, server outage (base station server down) results in 22.6% of packet losses. Then with *REFILL*, we find the causes for other packet losses. In those packet losses, duplicated losses, timeout losses and overflow losses are of 0.3%, 0.8% and 1.1%, respectively.

According to the output of *REFILL*, 32.2% of the losses are due to received loss. We further examine those received loss and find among those received losses, 20.0% are lost on the sink node and 12.2% are lost on other nodes.

An acked loss on node $n$ means that an acked packet still gets lost. 38.6% of losses are due to acked loss. Among those acked losses, 38.0% are acked loss on the sink node, i.e., nodes received packets from the sink node but the packets are lost. 0.6% are lost on other nodes.

### D. Implications

*1) Whose packets are lost and where packets are lost?:* We can calculate whose packets are lost from the collected data packets. As shown in Figure 4, it seems that different nodes have similar probabilities for lost packets and nodes play the same role in the network. For system designers, it seems that packet losses are evenly distributed in the network. However, by examining where packets are lost with *REFILL* on the collected log, we find that different nodes are significantly different. There exist a small portion of nodes where a large portion of packets are lost. Those nodes are much more important than other nodes in terms of packet losses. We also find that node location plays an important role. Though most routing protocols have considered node and link dynamics, node positions and connections should be carefully considered especially during the deployment phase. This can significantly improve the system performance.

*2) Correlation based approaches in time domain.:* Many existing works derive the packet loss causes using correlation based method in time domain [15]. To find the causes of packet losses, packet losses are correlated with events during the same time period. However, correlation based approach may have some limitations according to the result of *REFILL*. First, at the same time period, there may exist different causes. It is difficult to distinguish those causes at the same period, making correlation based method difficult to reveal the real cause. Even when there are only a single cause during a time period, correlation based approaches may still have some limitation. Some important causes (e.g., timeout loss) may only result in a few packet losses. Those packet losses are much less than that resulted from other causes and are easy to be overlooked.

*3) Node loss vs. link loss.:* Traditionally, we focus on link losses. From the result, we can see packet losses due to retransmission timeout are not high. For example, with up to 30 retransmissions for each packet, packet losses due to low link quality become very low. On the other hand, we find that many packets are lost even though they are successfully received at some node. This tells us that we should carefully examine problems inside each node. First, packet are delivered

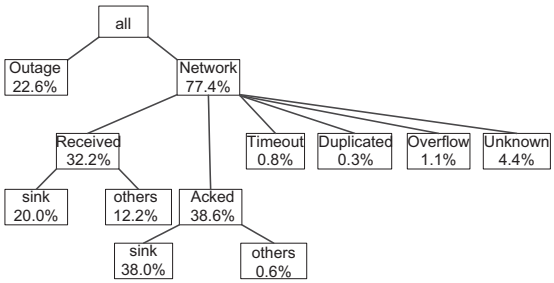


Fig. 9: Percentage of different causes.

from the low layer to the high layer in the operating system on sensor nodes. Due to limited resource, the OS often has some tradeoffs. For example, a task cannot be put into the queue when there is a same task in the queue. This may result in task failure and hence packet loss. Second, different components have different design objectives and those design objectives may even have conflicts. For example, in the radio communication component, a node will check the CCA value after disabling the interrupt. Packets received during such a time period may get lost.

*4) The last mile in the network.:* During the operational time, we have experienced many server outage events. Those events, which are not problems of WSNs, indeed result in packet losses. Before the deployment phase, we conduct extensive lab-tests and small outdoor scale tests. We focus on the performance of WSNs while overlook the backbone network to the base station and the connection from WSNs to the backbone network. This tells us that we should thoroughly test every part of the entire network.

*5) ACK mechanism.:* We find that a portion of packets may get lost even when ACKs for those packets are received at the sender. In our network, hardware ACK is used. The receiver delivers a packet to upper layers after ACK is sent at the PHY layer. However, this packet may not be successfully delivered to upper layers due to limited memory, computation resource, etc. For example, the memory may be full and not be able to accommodate a new packet or the MCU is too busy to process a new packet. Thus the packet will be discarded while being delivered to upper layers even it is received by the hardware. This also means a packet may still get "lost" even when the sender has successfully received the hardware ACK. An alternative approach is to send ACK at the software layer, i.e., send ACK after upper layers have successfully received the packet. However, this will introduce delay for the ACK, which decreases the transmission efficiency.

## VI. Related Work

There are a large collection of log analysis works. For example, Sherlog [17] is proposed to analyze local program log for diagnosis. In Sherlog, program logs are analyzed in order to investigate possible bugs in the program. Further, proactive logging method [19] [18] is proposed to improve the quality of logging. With proactive logging, logs can be appropriately inserted into the source code to facilitate the diagnosis in the program. Those works mainly consider using

the log on a single node. They do not use logs from multiple nodes in distributed networks to derive useful information. Our work is inspired by the work Wit [10] and NetCheck [21]. To study the detailed MAC layer behavior, Wit proposes a data analysis method based on wireless sniff data from different nodes. However, the method is based on sniffer data in which a portion of packets can be recorded on multiple sniffers. Logs are combined with common recorded events. When common events are lost or not recorded, logs cannot be combined. Implicit network correlations, which provide useful information to combine different logs, are not considered and leveraged. While in *REFILL*, we do not have commonly recorded synchronization events. NetCheck proposes a method to analyze the log data in the network. Event correlation is not considered while processing log. NetCheck does not show how to connect inference engines on different nodes and does not consider the impact of lost events.

There are also many works for path tracing by using data from different nodes. PathZip [9] presents a method to recover the path of the packet from the collected data. For example, PathZip uses a hashtable to store the nodes on the path. It is based on a precondition that neighboring nodes of each node are known in prior. Then it searches in each node's neighboring nodes to find nodes on the path hop by hop. Recently, DTrack [2] proposes a method for accurate path tracking with probings. DTrack improves existing methods since most existing works consider all paths equally. DTrack optimizes the probing in packet tracing according to the likehood of path changes. There are other packet tracing methods in Internet to improve the performance in trace route [12] [13]. There are also other works to trace the evolution of IP topologies [3] [7]. Different from those works, *REFILL* can recover the event flow and thus the packet path based on individual logs from different nodes in the presence of event losses.

## VII. Conclusions and Future Work

We present *REFILL*, a method to reconstruct network behavior with individual and lossy logs in distributed networks. The main idea of *REFILL* is to build inference engines for each node and connect multiple inference engines of different nodes with implicit correlations. *REFILL* can derive the event flow and thus reconstruct the network behavior from unsynchronized and incomplete logs. We apply *REFILL* to a real wireless sensor network project consisting of 1200 nodes. The results show that *REFILL* provides event flows that are otherwise difficult to achieve with other approaches. Based on the event flow, we can reveal the spatial and temporal properties of packets losses. We can also find the causes of packet losses. Further, we improve the network performance based on the results from *REFILL*. In the future, we will enhance *REFILL* to include more events in the network, and work on more efficient and effective logging methods for *REFILL*.

## References


[1] D. Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of ACM MobiCom*, 2003.

[2] I. Cunha, R. Teixeira, D. Veitch, and C. Diot. Predicting and tracking internet path changes. In *Proceedings of ACM SIGCOMM*, 2011.

[3] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *Proceedings ACM SIGMETRICS*, 2005.

[4] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proceedings of ACM SenSys*, 2009.

[5] M. Keller, J. Beutel, and L. Thiele. How was your journey? uncovering routing dynamics in deployed sensor networks with multi-hop network tomography. In *Proceedings of ACM SenSys*, 2012.

[6] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from tinyos programs using symbolic execution. In *Proceedings of IEEE/ACM IPSN*, 2008.

[7] M. Latapy, C. Magnien, and F. Oudraogo. A radar for the internet. In *Proceedings of Intl. Workshop on Analysis of Dynamic Networks*, 2008.

[8] M. Lee, S. Goldberg, R. R. Kompella, and G. Varghese. Fine-grained latency and loss measurements in the presence of reordering. In *Proceedings of ACM SIGMETRICS*, 2011.

[9] S. Li, X. Liao, D. Dong, and X. Lu. Pathzip: Packet path tracing in wireless sensor networks. In *Proceedings of IEEE MASS*, 2012.

[10] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing the mac-level behavior of wireless networks in the wild. In *Proceedings of SIGCOMM*, 2006.

[11] X. Mao, X. Miao, Y. He, X.-Y. Li, and Y. Liu. Citysee: Urban co2 monitoring with sensors. In *Proceedings of IEEE INFOCOM*, pages 1611–1619, 2012.

[12] R. Sherwood, A. Bender, and N. Spring. Discarte: a disjunctive internet cartographer. In *Proceedings of ACM Sigcomm*, 2008.

[13] N. Spring, R. Mahajan, and D. Wetherall. Measuring isp topologies with rocketfuel. In *Proceedings of ACM Sigcomm*, 2002.

[14] K. Srinivasan, P. Dutta, A. Tavakoli, and P. Levis. Understanding the Causes of Packet Delivery Success and Failure in Dense Wireless Sensor Networks (Technical report SING-06-00). Technical report, Stanford University, 2006.

[15] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California Fault Lines: Understanding the Causes and Impact of Network Failures. In *Proceedings of ACM SIGCOMM*, 2010.

[16] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Networks. In *Proceedings of USENIX OSDI*, 2006.

[17] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[18] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, Y. Zhou, and S. Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[19] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proceedings of the International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[20] J. Zhao and R. Govindan. Understanding Packet Delivery Performance In Dense Wireless Sensor Networks. In *Proceedings of ACM SenSys*, 2003.

[21] Y. Zhuang, E. Gessiou, S. Portzer, F. Fund, M. Muhammad, I. Beschastnikh, and J. Cappos. Netcheck: Network diagnoses from blackbox traces. In *Proceedings of USENIX NSDI*, 2014.