Error Scene Restoration with Runtime Logs of Wireless Sensor Networks

Shuo Lian*, Xiuzhen Guo†, Zhenge Guo‡, Xu Zhao§
*Huawei Technologies Co. Ltd.
†School of Software and TNLIST, Tsinghua University
‡Xi'an Jiaotong University
§Xi'an Thermal Power Research Institute Co. Ltd.
guoxz16@mails.tsinghua.edu.cn

***Abstract*—The application of Wireless Sensor Networks (WSNs) often falls into unexpected poor performance conditions due to many factors such as complex network interactions, software bugs and incorrect configurations. Diagnosing such a network is challenging since it is difficult to obtain information from the network due to factors including (1) non-deterministic network interactions among motes; (2) difficulties in reconstructing the status of each individual mote; and (3) unavailability of the real environment information. To address these problems, we propose a diagnosis tool called ALog which analyzes the local logs and the source code to infer what happens in network. Based on the analysis, we further derive the states and possible problems accordingly. We implement ALog and evaluate its efficiency with two real case studies. The results demonstrate that ALog is accurate and applicable for diagnosing real sensor networks.**

I. Introduction

The applications of Wireless Sensor Networks (WSNs) require high availability and reliability in real world[1–4]. When the WSN system faces unexpected behaviors, e.g.poor network performance or partial unknown failures, administrator and service engineers are required to diagnose and solve the problem efficiently. However, diagnosing a WSN is challenging with limited evidences including: network logs collected from motes, application source code and deployment topology. Furthermore, there are various reasons to obscure the administrator to get the fact about what really happened in network such as non-deterministic behaviors in network interaction among motes, difficulty in reproducing the real preemptive execution paths in mote and unavailability of the real information of environments.

Existing works in the area of WSNs can partially resolve the above issue. Specifically, offline analysis based approaches [5] collect certain network traces and then offer the functionality of trace based simulations and tests. Those approaches are exhaustive methods and perform plenty of tests to WSN applications, Unfortunately they are likely to miss the real informative evidences for the network. Another category of approaches are based on runtime diagnosis [6] analysis. Due to the complexity of network interactions and software processing, those approaches usually have to inject numerous tracepoints or agents into the node program, possibly incurring excessive costs in communication, memory, and computation. As a summary, it is important to provide mote information and error scene during the suspicious time, however, such a tool is still not available.

The comprehensive error scene for a problematic network should be considered in two aspects: (1) the error scene should be related to the source code, so as to help find out the detailed behavior in network interactions. (2) The network evidences should be considered in order to narrow down the probable cause. However there are difficulties in both how to model the interactions and how to leverage the network log inferring the error scene.

In order to address the above issues, in this paper we propose ALog, a diagnostic approach for error scene restoration in WSNs. ALog utilizes the logs recorded on sensor nodes to retrieve runtime network information and conducts offline analysis to restore the error scenes. It models the software execution with distributed preemptive features as a Network Context Control Flow Graph (NCCFG) for WSNs. Based on NCCFG, ALog infers all possible pathlets that connect every two adjacent log entries under certain variable constraints. By merging all those deterministic and possible pathlets, ALog is able to restore all scenes that possibly took place during the interested operational period. We propose a two-step log-driven inference algorithm to restore the error scenes and implement ALog based on TinyOS and present two real cases as well as evaluation results to demonstrate the effectiveness of ALog.

The remainder of this paper is organized as follows. We elaborate on the design of ALog in Section II, followed by evaluation and case studies in Section III. We discuss the related work in Section IV and conclude this work in Section V.

II. ALog System Design

This section introduces the design of ALog. We start with an overview of ALog. Then we respectively elaborate on the two core steps: network log parsing, network behaviors and internal states inference.

A. ALog Overview

Generally, ALog is designed for inferring and reconstructing network error scene to help network administrator understand what have happened in WSNs during the poor performance condition.

Types of Network Scenes: According to the complex network interaction behaviors and the preemptive execution model of nesC language, inferring and reconstructing network error scene form network logs is a significant challenge. In this work, we leverage both the network logs information and the rule of network interaction hidden in mote source code and

978-1-5090-5696-5/16 $31.00 © 2016 IEEE
DOI 10.1109/MSN.2016.57

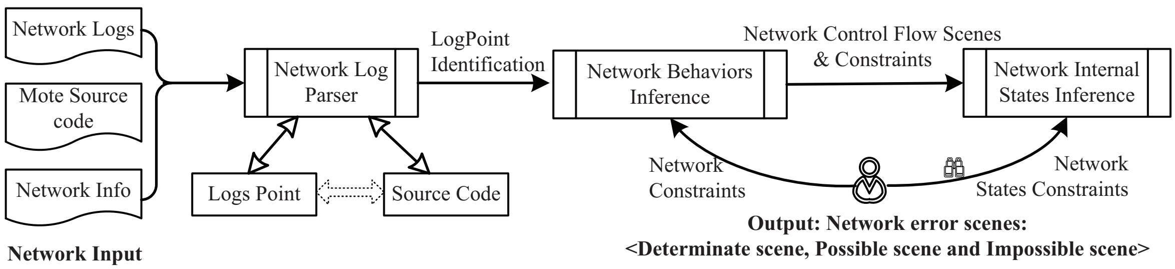

Network Logs
Mote Source code
Network Info
Network Input
Network Log Parser
LogPoint Identification
Logs Point
Source Code
Network Behaviors Inference
Network Control Flow Scenes & Constraints
Network Internal States Inference
Network Constraints
Network States Constraints
Output: Network error scenes:
<Determinate scene, Possible scene and Impossible scene>

Fig. 1. The workflow of ALog.

nesC execution model to narrow down the possible network scenes. Basically, we cut output network scenes into three parts include: determinate Scene, possible scene and impossible scene.

Determinate Scene: the partial interaction processes that definitely have occurred and mote internal states (a set of variables values) were held during the unexpected condition.

Possible Scene: the partial interaction process that may have occurred and mote internal states (a set of variables values) may have been held during the unexpected condition.

Impossible Scene: the partial interaction processes that have definitely not occurred under the corresponding constraints during the unexpected condition.

To accomplish above objectives, ALog first needs to parse network logs and using these logs to identify starting points in source code for information inference. Then using the initial information provided by logs, it tries to statically "walk" through the code to infer the above information based on the Network Context Control Flow Graph (NCCFG) that maps the nesC code execution model and the rules of network interaction. The workflow of ALog is shown as in figure 1.

B. Network Log Parsing

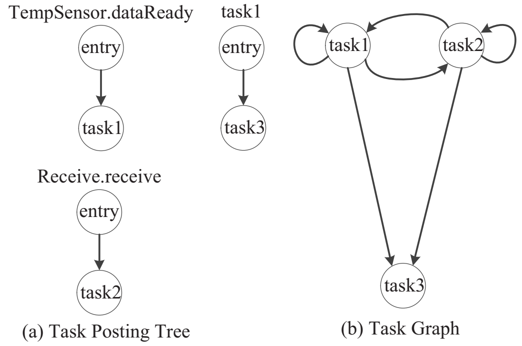
The basic version of this parsing process is a string matching work if all log messages are simply produced by statements like "*printf*". We could use the regular expression to match the variable values in log message.

However, for a modularized logging facility which has complicated wrappers to support customized logging format, the basic parsing process could not work well because the real string of a log statement is hidden in log wrappers. We simply discuss the solution as extension. Our intensive parsing process could handle this problem by tracing log message in Control Flow Graph (CFG). The real string of one log statement would be achieved by this way.

C. Network Behaviors Inference

To guarantee the ALog practical and accurate for a real sensor network system, we design this part into two steps:

1) ALog first constructs a Network Context Control Flow Graph (NCCFG) for network.
2) ALog starts on searching possible pathlets under corresponding constraints between two adjacent logging messages of the subsequence network logging file in NCCFG. Then ALog uses two algorithms to print all network scenes as results which are classified as three parts: determinate scene, possible scene and impossible scene.



(a) Task Posting Tree (b) Task Graph

Fig. 2. An example of Task Posting Tree and the corresponding Task Graph.

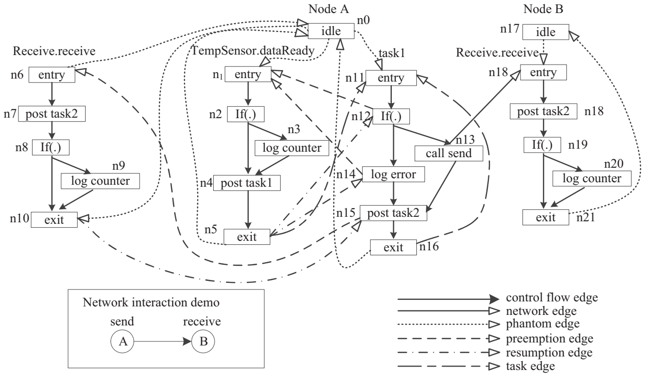


Fig. 3. An example of partial NCCFG for network scenes. Note that this figure omits some edges for clarity representation such as preemption edge, resumption edge and control flow edge.

Definition 1: A Task Posting Tree (TPT) for a module m is a directed tree of 2-tuple $G_{TPT}(m) = (V_m, E_m)$, where V_m is a set of vertexes and E_m is a set of directed edges. Each vertex is a post statement in control flow graph.

Definition 2: A Task Graph (TG) of a module m is denoted as a directed graph of 2-tuple $G_{TG}(m) = (V_m, E_m)$, where V_m is a set of task components in m, and E_m is set of directed edges indicating possible execution sequences among these candidate tasks. If there is an edge $e \in E_m < V_i, V_j >$, we say task V_j has opportunity to execute after task V_i.

Figure 2 is a demo shows the Task Posting Tree and the corresponding Task Graph for the motivating example

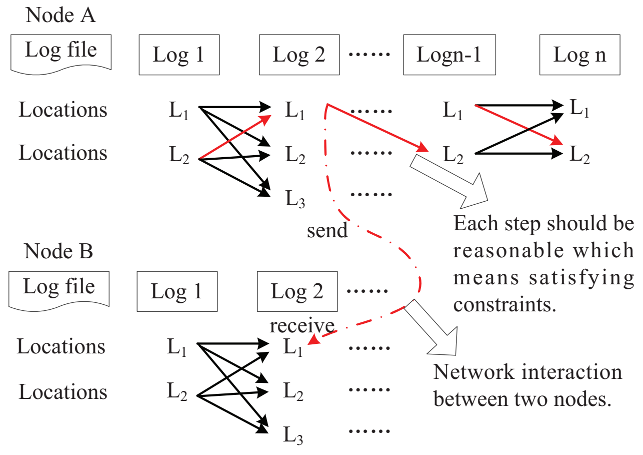


Fig. 4. Log-driven matching process in ALog.

in section II. In motivating example, there are two event handlers $Receive.receive$ and $TempSensor.dataReady$ post $task2$ and $task1$ respectively. This type of structure could be modeled as $entry \rightarrow task\#$ shown as in figure 2(a). Other type of structure describes the relationship among tasks.

Furthermore, we introduce a new network context control flow graph, called NCCFG, to express the interaction among motes based on TG. An NCCFG of a network n aims to express the preemptive execution behavior of network caused by interrupts, deferred execution of tasks and network interaction information. The definition is as follows.

Definition 3: A Network Context Control Flow Graph (NCCFG) of a sensor network N is defined as a directed graph of 6-tuple $G_{NCCFG}(N) = (G_{CFG}, E_{network}, E_{phantom}, E_{preemption}, E_{resumption}, E_{task})$, where G_{CFG} are control flow graph of all modules in network. $E_{network}$,$E_{phantom}$,$E_{preemption}$,$E_{resumption}$,E_{task} denote network edges, phantom edges, preemption edges, resumption edges and task edges respectively.

In order to illuminate the detail design of NCCFG, we use a concrete example to express the necessary information with each type of edge shown in figure 3.

The next challenge focuses on how to use the network logs to narrow down the possibility of network scenes. Based on NCCFG, our approach proposes a log-driven matching algorithm that leverages 5 special rules to output the possible network scenes efficiently.

First ALog gets the all reasonable possible pathlets between two adjacent logs which should satisfy the corresponding constraints, for example if there is a path outputs log $m1@line\#07$ and $n1@line\#30$, sub-procedures $task1@line\#05$ and $dataReady@line\#28$ must under the constraints $c1$($highTemp = true$ and someone have post $task1$ before)and $c2$ ($verbose = true$ and temperature sensor data have ready) respectively in motivating example. Figure 4 illustrates the step about each two adjacent logs matching process. Note those network logs are connected by network interaction for example send and receive event pairs. The detail algorithm of this sub-step is shown in Algorithm 1.

Algorithm 1 $getAdjacentLogPath(G_{NCCFG}, v)$.

```
Input:
    $G_{NCCFG}(N)$: $N$ indicates the abstraction of network and
    $G_{NCCFG}$ denotes the NCCFG
Output:
    $P_{sub}(p, c)$: A set of possible pathlets $p$ and corresponding
    constraints $c$ between two adjacent logs.
mark $v$ as explored, combine the sub-path constraints $c$
for all edges $e$ in $G_{NCCFG}(N).AdjacentEdges(v)$ do
  if edge $e$ is unexplored and $reasonable(e)$ then
    $v^* \leftarrow G_{NCCFG}(v, e).AdjacentVertex(v, e)$.
    if vertex $v^*$ is unexplored then
      mark $e$ as discovery edge.
      add $v^*$ and merge $c^* = getConstraints(v, v^*)$ to
      $P_{sub}$.
      update context to $reasonable(e)$.
      recursively $getAdjacentLogPath(G_{NCCFG}, v^*)$.
    else
      label $e$ as a back edge.
    end if
  end if
end for
return $P_{sub}$;
```

Algorithm 2 $getNetworkScenes(G_{NCCFG}, v)$.

```
Input:
    $P_{sub}(p, c)$: A set of possible pathlets $p$ and corresponding
    constraints $c$ between two adjacent logs.
Output:
    $NS(p, c)$: A set of possible network scenes and corre-
    sponding constraints $c$.
merge all pathlets in $P_{sub}$ as an entire network scenes
($NS$) based on the satisfiability $c1 \wedge c2 \wedge c3 \wedge ...$ in network
scene.
if SAT not satisfied ($c1 \wedge c2 \wedge c3 \wedge ...$==false) then
  mark as impossible scene in $NS$.
end if
if SAT satisfied ($c1 \wedge c2 \wedge c3 \wedge ...$==true)and all network
scenes use this pathlet then
  mark as determinate scene in $NS$.
else
  mark as possible scene in $NS$.
end if
return $NS$;
```

III. Evaluation and Case Studies

This section demonstrates the effectiveness of ALog as a diagnosis tool and two real case studies.

A. Methodology

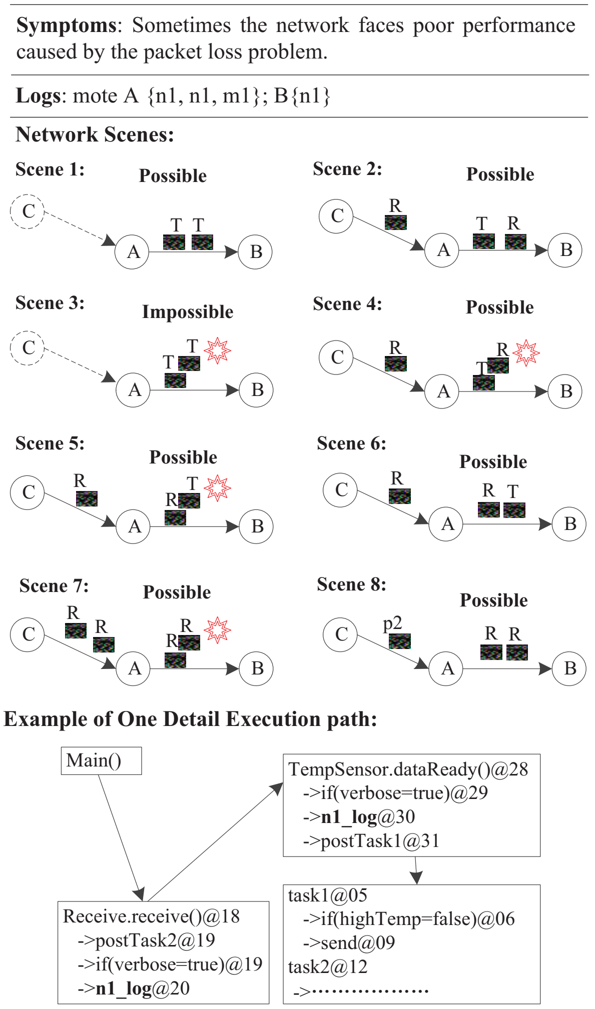
We evaluate the performance of NCCFG model on real world applications adapted from the TinyOS distribution and CitySee. Furthermore we show two cases study how Alog help a programmer to verify the hypothesis he/she holds based on network logs.

B. Case 1: A Race Failure about Share Variable

For a long running system, the administrator always face maintain and diagnosis problem. Figure 5 shows a simplified

TABLE I. NCCFG DETAIL INFORMATION FOR DIAGNOSING BY ALOG.

Applications	# of vertex	TG	LOC	log points	Pathlet length
Blink	17	3	132	4	3
TestNetwork	127	0	351	7	5.5
TestNetworkLpl	129	0	353	7	5
CitySee	129	0	4679	20	7



Symptoms: Sometimes the network faces poor performance caused by the packet loss problem.

Logs: mote A {n1, n1, m1}; B{n1}

Network Scenes:

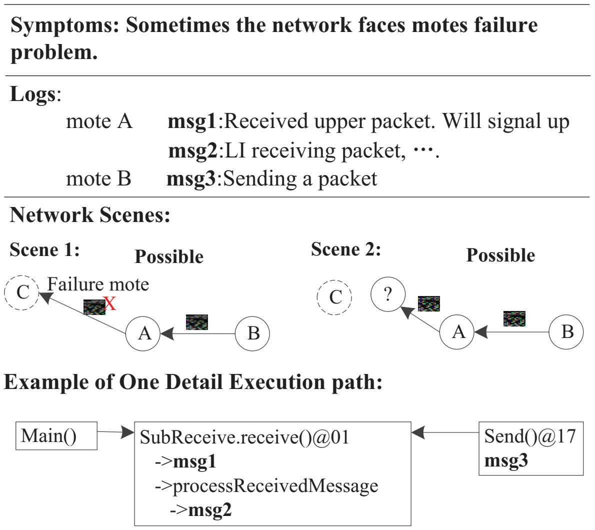
Example of One Detail Execution path:

```
Main()

TempSensor.dataReady()@28
  ->if(verbose=true)@29
  ->n1_log@30
  ->postTask1@31

Receive.receive()@18
  ->postTask2@19
  ->if(verbose=true)@19
  ->n1_log@20

task1@05
  ->if(highTemp=false)@06
  ->send@09
task2@12
 ->..................
```

Fig. 5. Diagnosis report for case 1 by ALog.

ALog report for this problem includes: symptoms, possible network scenes and so on. For clear demonstration, we simply list 8 different network scenes from more than 64 possible scenes produced by ALog and mark them as determinate scene, possible scene and impossible scene to help diagnosis.

We explain the scenes as follows. For scenes $1, 2, 6$ and 8, the packet loss problem is caused by poor link. Furthermore in spite of link quality or collision problem, there still is possibility lead to failure of sending operation in mote A, such as scenes $3, 4, 5$, and 7. These scenes reveal a potential bug in source code, which is caused by two preemptive event handlers share a variable $smsg$ without considering data race. So the share variable $smsg$ could be re-written between first send event and $senddone$ event. The execution order also could cause sending failure error by two continuous send calls without $senddone$ event for example scenes $4, 5$ and 7. Note that scene 3 is an impossible scene which breaks constrain of periodical timer because the logic of design guarantee temperature sensing operation would not continuously invoked quickly.



Symptoms: Sometimes the network faces motes failure problem.

Logs:
mote A **msg1**:Received upper packet. Will signal up
msg2:LI receiving packet, ⋯.
mote B **msg3**:Sending a packet

Network Scenes:

Example of One Detail Execution path:

```
Main()

SubReceive.receive()@01
  ->msg1
  ->processReceivedMessage
    ->msg2

Send()@17
msg3
```

Fig. 6. Diagnosis report for CitySee by ALog.

```
#01:  event message_t* SubReceive.receive(message_t* msg,
#02:                                   void* payload, uint8_t len) {
#03:   log("LI", "Received upper packet. Will signal up\n");//msg1
#04:   processReceivedMessage(msg, payload, len);
#05:   return signal Receive.receive(msg, call Packet.getPayload(msg,
#06:                        call Packet.payloadLength(msg)),
#07:                                call Packet.payloadLength(msg));
#08:  }
#09:  void processReceivedMessage(message_t* ONE msg, void* (len)
#10:                               payload, uint8_t len) {
#11     uint16_t id2replace;
#12:   id2replace = AM_BROADCAST_ADDR;
#13:   //Fixing: add a timer to evict dumb motes
#14:   log("LI", "receiving packet, buf addr: %x\n", payload);//msg2
#15:  }
#16:    // code omitted
#17:  log("sending packet, buf addr: %x\n", payload);//msg3
#18:  call Send.send(&msg, sizeof(msg));
```

Fig. 7. Code Link Estimator Bug in CitySee.

C. Case 2: A Routing Error about CTP

This case introduces a real diagnosis problem form CitySee system, a large-scale WSN system in urban. In earliest period of CitySee, we adopt collection tree protocol (CTP) as the basic routing protocol. However sometimes the administrator finds a performance problem shows some motes seems failure and could not report sensing data to base station. Then the administrator achieves the corresponding network logs collected from motes and uses ALog to infer the possible network scenes based on logs and source code. Figure 6 shows a simplified report produced by ALog and Figure 7 shows the corresponding source code.

IV. Related Work

Existing diagnosis system approaches for WSNs, based on log or trace, fall into two categories: before or after deployment.

The first category always leverages testing or simulating technique to troubleshoot the local error in mote at before deployment. Safe TinyOS is a diagnosing tool for checking type-safe feature of application in memory. It is embedded in TinyOS compilation toolchain and warns the developer about unsafe code fragments during compilation and additionally instruments the assertion code with safety annotations preventing memory corruption at runtime. Other typical approach is KleeNet [7], which is a debug environment running configured sensor network programs on symbolic input and automatically tests non-deterministic possible failures. In this way, program developers are able to discover potential bugs and verify the reliability of application before deployments. Other related works include T-check [5], FSMGen [8] and so on. However, these approaches often provides simulating and testing function based on application source code, but fail to solve a real problem based on network logs information.

The second category mainly focuses on the real log collected from real system on analyzing the possible error in mote. DustMiner [9] and LiveNet [6] collect informative messages from the sensor network for fine-grained visibility into network interactions and operations without requiring much resource in mote. EnviroLog [10] records some events described nondeterministic behaviors for producing efficient in-network execution replay. Sympathy [11] only collects a small amount of diagnosis data to identify the root cause of network failures. Marionette and Clairvoyant [12] provide remote debugging access interface to source-level symbols and statements in the source code of programs on the sensor motes. Declarative Tracepoints provide a programmable SQL-like interface to describe customizable debugging operations. The associated debugging information could be disseminated and executed on the remote motes at runtime, enabling convenient remote debugging. But these approaches usually needs excessive agent injected in motes and consumes communication bandwidth, but could not combine network logs information and source code to diagnose problematic network.

V. Conclusion

Wireless sensor networks are deemed as an affordable solution to provide sustainable and efficient sensing services in the real world. When a system of WSNs faces unexpected condition for example poor network performance, administrator and service engineers are required to quickly diagnose and solve the problem. For administrator and service engineers, unfortunately, it is difficult to achieve informative data in lab about what the network had real happened. To address this problem, we propose ALog, a diagnostic approach for error scene restoration in WSNs. ALog combines the logs collected from motes and source code to retrieve runtime network information and conducts offline analysis to restore the error scenes based on NCCFG. The report of ALog includes network scenes marked as determinate scene, possible scene and impossible scene by checking corresponding constraints. We implement ALog and evaluate its efficiency with two real cases studies. The results demonstrate that ALog is accurate and applicable to a real sensor networks.

VI. Acknowledgments

This research is supported in part by the National Natural Science Fund China under Grant No.61373146 and No.61472382.

References

[1] X. Zheng, Z. Cao, J. Wang, Y. He, and Y. Liu, “Zisense: towards interference resilient duty cycling in wireless sensor networks,” in *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*. ACM, 2014, pp. 119–133.

[2] D. Liu, M. Hou, Z. Cao, J. Wang, Y. He, and Y. Liu, “Duplicate detectable opportunistic forwarding in duty-cycled wireless sensor networks,” *IEEE/ACM Transactions on Networking*, vol. 24, no. 2, pp. 662–673, 2016.

[3] Y. He, H. Ren, Y. Liu, and B. Yang, “On the reliability of large-scale distributed systems–a topological view,” *Computer networks*, vol. 53, no. 12, pp. 2140–2152, 2009.

[4] L. Wang, X. Qi, J. Xiao, K. Wu, M. Hamdi, and Q. Zhang, “Exploring smart pilot for wireless rate adaptation,” 2016.

[5] P. Li and J. Regehr, “T-check: bug finding for sensor networks,” in *Proceedings of the 9th IEEE/ACM IPSN*, 2010.

[6] B. Chen, G. Peterson, G. Mainland, and M. Welsh, “Livenet: Using passive monitoring to reconstruct sensor network dynamics,” in *Proceedings of the 4th IEEE DCOSS*, 2008.

[7] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, “Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment,” in *Proceedings of the 9th IEEE/ACM IPSN*, 2010.

[8] N. Kothari, T. Millstein, and R. Govindan, “Deriving state machines from tinyos programs using symbolic execution,” in *Proceedings of the 7th IEEE/ACM IPSN*, 2008.

[9] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han, “Dustminer: troubleshooting interactive complexity bugs in sensor networks,” in *Proceedings of the 6th ACM SenSys*, 2008.

[10] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic, “Achieving repeatability of asynchronous events in wireless sensor networks with envirolog,” in *Proceedings of the 25th IEEE INFOCOM*, 2006.

[11] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, “Sympathy for the sensor network debugger,” in *Proceedings of the 3rd ACM SenSys*, 2005.

[12] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse, “Clairvoyant: a comprehensive source-level debugger for wireless sensor networks,” in *Proceedings of the 5th ACM SenSys*, 2007.