This paper was presented as part of the main technical program at IEEE INFOCOM 2011

Self-Diagnosis for Large Scale Wireless Sensor Networks

Kebin Liu[1,2], Qiang Ma[2], Xibin Zhao[1] and Yunhao Liu[1,2]

1 TNLIST, School of Software, Tsinghua University

2 Hong Kong University of Science and Technology

{kebin, maq}@cse.ust.hk, zxb@tsinghua.edu.cn, liu@cse.ust.hk

***Abstract*—Existing approaches to diagnosing sensor networks are generally sink-based, which rely on actively pulling state information from all sensor nodes so as to conduct centralized analysis. However, the sink-based diagnosis tools incur huge communication overhead to the traffic sensitive sensor networks. Also, due to the unreliable wireless communications, sink often obtains incomplete and sometimes suspicious information, leading to highly inaccurate judgments. Even worse, we observe that it is always more difficult to obtain state information from the problematic or critical regions. To address the above issues, we present the concept of self-diagnosis, which encourages each single sensor to join the fault decision process. We design a series of novel fault detectors through which multiple nodes can cooperate with each other in a diagnosis task. The fault detectors encode the diagnosis process to state transitions. Each sensor can participate in the fault diagnosis by transiting the detector's current state to a new one based on local evidences and then pass the fault detector to other nodes. Having sufficient evidences, the fault detector achieves the *Accept* state and outputs the final diagnosis report. We examine the performance of our self-diagnosis tool called TinyD2 on a 100 nodes testbed.**

I. INTRODUCTION

Wireless sensor networks (WSNs) that enable many surveillance applications [11, 16, 27, 29] usually consist of a large number of sensor nodes deployed in the wild. Accurate and real-time fault diagnosis is of great significance for ensuring the system functionality and reliability. A few faulty nodes can heavily degrade the system performance and shorten the network lifetime. Nevertheless, diagnosis for a large scale and in-situ WSN is quite challenging. As the network size increases, the interactions among sensors within a network are extremely complicated. Meanwhile, the ad hoc manner of WSNs prevent network administrators from looking into the in-network structures and node behaviors, so it is difficult to pinpoint the root causes when abnormal symptoms are observed.

This work is motivated from our ongoing forest surveillance project GreenOrbs [20] with up to 330 sensor nodes deployed in the wild. GreenOrbs enables several applications such as canopy closure estimates and atmospheric concentrations of carbon dioxide in the forest. Figure 1 shows a topology snapshot of the GreenOrbs system. During the 10 months deployment period, we experience frequent abnormal symptoms in the network such as high data loss, temporary disconnection of nodes in a certain region, rapid energy depletion of some nodes and pervasive sensing errors. Trouble-shooting root causes in such a large network is crucial while time consuming, so efficient and accurate fault diagnosis tools are necessary for ensuring the system performance and lifetime.

Existing approaches to diagnosing WSNs are mainly sink-based. They often rely on "pulling" state information from all nodes and conducting fault inference process at the back-end. For example, Sympathy [22] periodically collects status information from sensor nodes such as connectivity metrics and flow metrics. According to these metrics, sink trouble-shoots the root-causes with a decision tree. Zhao *et al.* [30] propose the energy scan scheme which monitors the residual energy and other parameter aggregates on sensor nodes and reports to the sink. Those approaches are not feasible in a large scale sensor network due to the following two major shortages. On the one hand, proactively information leads to large communication overhead that heavily shortens the system lifetime. Normally, errors affect a portion of nodes in the problematic regions, so we only need to derive state information from the particular group of nodes for root-causes discovery. Frequent information collection from all sensors is not necessary and actually wastes a lot of energy. On the other hand, due to the large network scale and the unreliable wireless communications, the fault inference engine on the back-end generally ob-

978-1-4244-9920-5/11/$26.00 ©2011 IEEE

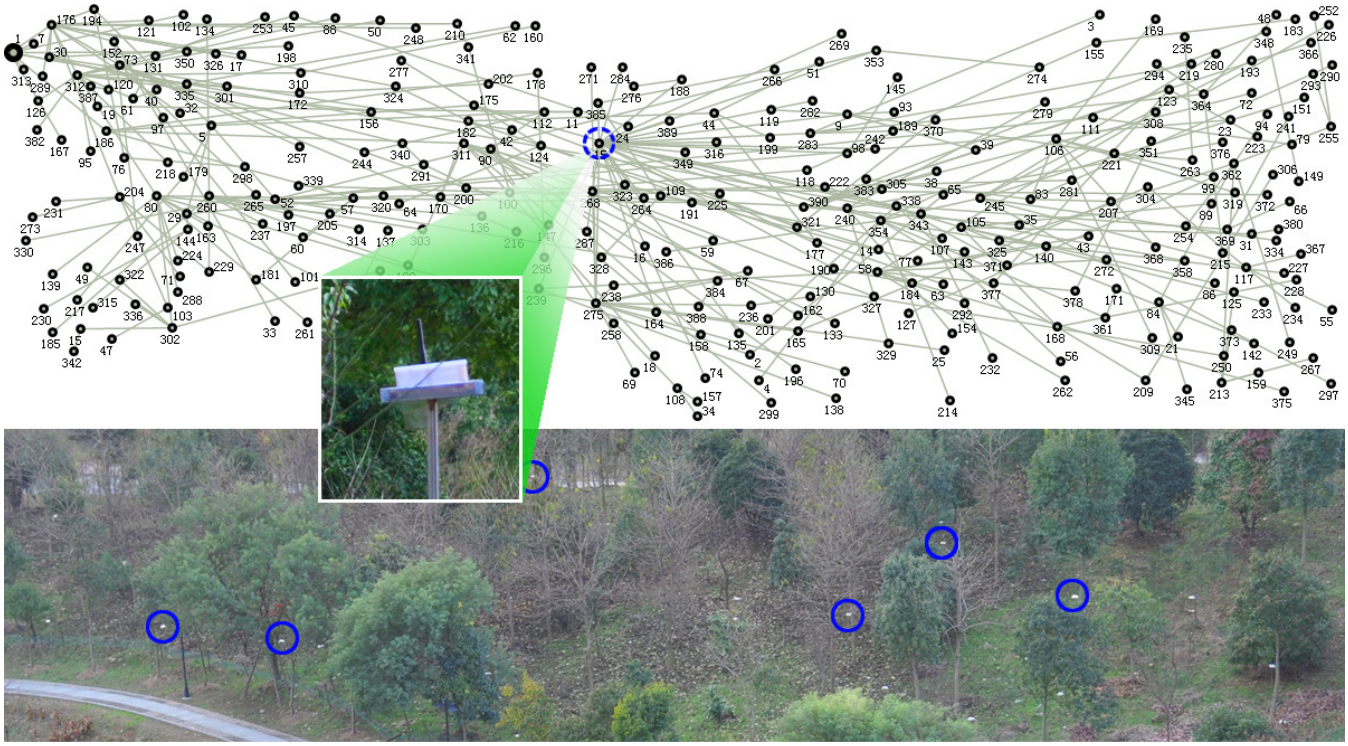

Figure 1 GreenOrbs forest surveillance project

tains incomplete and uncertain information which significantly deteriorates the diagnosis accuracy.

To address these issues, we propose to let every single sensor to be involved in the diagnosis process. We present the concept of self-diagnosis which enables sensors in a network to diagnose themselves. We "push" some diagnosis tasks to sensor nodes so they can join the diagnosis process and infer the root causes based on local evidences. The advantages of self-diagnosis are as threefold. First, self-diagnosis can save a large amount of transmissions by applying local decision. Second, it provides more real-time diagnosis results. Finally, self-diagnosis avoids information loss on the way to sink and thus improves the accuracy.

In spite of all these benefits, employing a self-diagnosis strategy in a large scale WSN is non-trivial and faces many difficulties. First, the computation and storage resources at each sensor are limited, so the components injected to nodes have to be light-weight. Second, a sensor only has narrow scope on the system state and in many cases it can hardly determine the root-causes simply based on local evidences. Hence how to make multiple sensors cooperate with each other to detect network failures needs to be addressed.

In this study, we present our new self-diagnosis tool named TinyD2. We introduce our fault detectors based on the Finite State Machines (FSMs). The fault detectors encode the diagnosis process to state transitions. Each fault detector has several states, each of which indicates an intermediate step of the fault diagnosis process. Taking the local information from each node as input, the fault detector switches its state. Thus, a sensor can participate in the diagnosis by transiting the state of the fault detector to a new one based on local evidences and then deliver the detector to other nodes. Fault detectors travel among sensor nodes and continuously collect new evidences. When a detector achieves its *Accept* state, the fault is pinpointed and a diagnosis report can be formed accordingly. During the fault inference process, sensor nodes cooperate with each other to achieve the final diagnosis report that can assist network administrators in conducting further diagnosis or failure recovery actions. The detectors are light-weight in terms of storage space and computation overhead. During the diagnosis process, messages exchanged among sensors include not full detectors but only the current state of them which is necessarily small in size. We also form the diagnosis trigger as a change-point detection problem and presents a triggering strategy based on the cumulative sum charts and a bootstrap scheme.

The major contributions of this work are as follows:

1) We study the problem of fault diagnosis in an operational sensor network and present a novel self-diagnosis framework for large scale WSNs.

2) We design fault detectors based on Finite State Machines, through which multiple nodes cooperate with each other and achieve diagnosis results.

3) We form the diagnosis triggering as a change-point detection problem and address it based on cumulative sum charts and a bootstrap scheme.

4) We Implement our self-diagnosis approach TinyD2 and conduct intensive experiments to examine its effectiveness in a 100 nodes testbed.

The rest of this paper is organized as follows. Section II introduces existing efforts that are related to this work. Section III describes the fault detectors and the self-diagnosis process. We discuss the implementation details of TinyD2 in Section IV, and present the experimental results in Section V. We conclude the work in Section VI.

II. Related Work

Both debugging and diagnosing approaches are related to this work. Debugging tools [17, 23] aim at finding software bugs in sensor nodes. Clairvoyant [28] enables the source-level debugging for WSNs which allows people remotely execute standard debugging commands such as step and break. Dustminer [14] focuses on troubleshooting interactive complexity bugs by mining the logs from sensor nodes. Declarative Tracepoints [5] provides the function of inserting tracepoints to applications at runtime and thus enabling developers to watch the values of certain variables. Debugging tools are powerful at finding network faults, however, in cost of incurring huge control overhead, so they are generally used in the pre-deployment scenarios.

However, a post-deployment WSN may experience many failures even when there is no bug in its software system. Diagnosis tools try to monitor the network state and trouble-shoot failures at real time. Some researchers propose to periodically scan parameters like residual energy [30] on sensor nodes. MintRoute [26] visualize the network topology by collecting neighbor tables from sensor nodes. SNMS [24] constructs network infrastructure for logging and retrieving state information at runtime and EmStar [8] supports simulation, emulation and visualization of operational sensor networks. LiveNet [6] provides a set of tools and techniques for reconstructing complex dynamics of live sensor networks. Other researchers employ varying inference strategies for trouble-shooting network faults. Sympathy [22] actively collects metrics such as data flow and neighbor table from sensor nodes and determines the root-causes based on a tree-like fault decision scheme. PAD [18] proposes the concept of passive diagnosis which leverages a packet marking strategy to derive network state and deduces the faults with a probabilistic inference model. Megan et al. [25] present Visibility, a new protocol design that aims at reducing the diagnosis overhead. PD2 [7] is a data-centric approach that tries to pinpoint the root-causes of application performance problems. Besides the functional failures, some researchers also tackle the problem of detecting faulty sensing data [10, 21]. Nevertheless, most of the existing diagnosis solutions are sink-based and in this work we propose a novel self-diagnosis approach which encourages sensor nodes to join the fault decision process.

Network diagnosis techniques for large scale enterprise networks and Internet are also related to this work. Commercial tools [1-3] are applied to monitoring the working status of servers and routers with various control messages. SCORE [15] and Shrink [12] diagnose the network failures based on shared risk modeling. Sherlock [4] presents a multi-level inference model that achieves high performance. Giza [19] try to address the performance diagnosis problem in a large IPTV network and NetMedic [13] enables detailed diagnosis in enterprise networks. Due to the ad hoc nature of sensor networks, enterprise network oriented approaches are infeasible for WSNs.

III. In-Network Fault Decision

In this section, we describe our self-diagnosis scheme for in-network fault decision. We propose a series of fault detectors that can act as glue and stick different sensors to a diagnosis task. After the system is deployed, each sensor continuously monitors its local system information. If some exceptions occur, the sensor creates a new fault detector. Detailed diagnosis triggering scheme will be discussed in Section III C. A fault detector has several states and each state denotes an intermediate diagnosis decision. Diagnosis is performed by state transition in the fault detector based on the input evidences.

During the diagnosis process, sensor nodes only transmit the current state of the fault detector and the detector structure is stored in each sensor. Once a final diagnosis decision is achieved, the sensor node will cache the report for a specified period. The report handling actions includes reporting the diagnosis decision to sink as well as start the active state collection components if more information is required by the sink.

A. *Fault Detector Design*

In TinyD2, the fault detectors encode the fault inference process to state transition. Our detectors are based on the Finite State Machines (FSMs) model. A FSM model consists of a finite number of states and transitions between these states. A transition in FSM means a state change which will be enabled when specified condition is fulfilled. Current state is determined by the historical states of the system, so it indicates the series of inputs to the system from the very beginning to present moment. Our fault detectors generalize the FSM model and use local evidences on each sensor node as inputs. Each state can be seen as an intermediate diagnosis decision and if the local evidences support certain conditions on current state we transit the detector to the corresponding new state. As one fault detector cannot cover all kinds of exceptions, we design various fault detectors for different abnormal symptoms. We consider three classes of symptoms. The first category of symptoms is caused by local errors such as the low battery power or system reboot, which means we can pinpoint the root

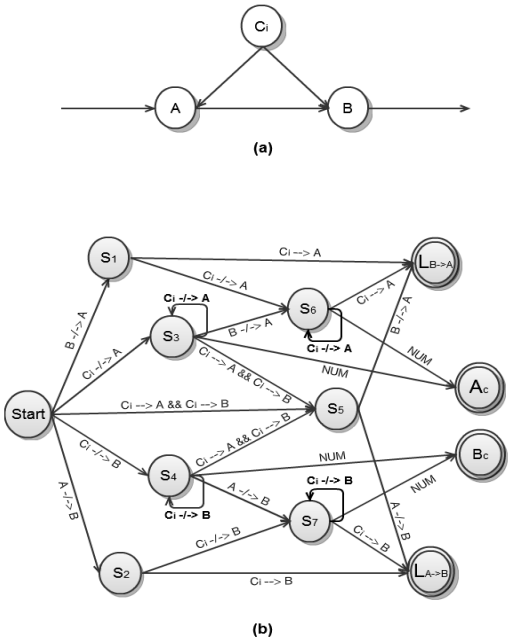

Figure 2 An example of the fault detector

causes from the local evidences only. The second category relates to failures on other nodes, for example if current node detects that a neighbor has just been removed from its neighbor table, it will issue a fault detector to neighborhood to make sure whether this neighbor is still alive. The third category of symptoms can be caused by local or external problems. For example, the node experiences a high retransmission ratio on its current link. However, it is not able to know whether this is because of the poor link quality or congestion at the receiver.

Now we take the high retransmission ratio as an example to describe our fault detector design. The fault detector for this exception is shown in Fig. 2.

In this example, we assume that sensor node *A* is transmitting packets to node *B*, { C_i } denotes a group of sensors in the neighboring region of *A* and *B* that will communicate with *A* and *B* as illustrated in Fig. 2 (a). At the very beginning, *A* detects that the retransmission ratio on link *A* -> *B* abruptly turns to be significantly high, so *A* creates a new fault detector as shown in Fig. 2 (b). Formally, the fault detector *M* is represented as a quintuple $M = (E, S, S_0, f, F)$ where *E* denotes the set of input evidences, *S* is the set of states in which S_0 is called the *Start* state, *F* is a subset of *S* that includes all *Accept* states, *f* is the state-transition function $f : S \times E -> S$ which specifies the conditions of state switching. In the example of Fig. 2 (b), cyclic vertices S_i denote the states in *S*, each arc indicates a possible transition from one state to another and annotations on the arcs specify the transition conditions. Note that some states may have self-loops. States denoted by double cycles are the *Accept* states which represent final diagnosis decisions. When a fault detector reaches an *Accept* state, we can conclude the fault type and root causes.

Now we describe the details of a diagnosis process with the detector shown in Fig. 3. Assume that *A* detects an abnormal retransmission ratio change on its link *A* -> *B*, it creates a new fault detector which is now at the *Start* state. In this detector, the *Start* state has the meaning that some node *A* finds that its transmission performance experiences problems on link *A* -> *B*. *A* then broadcasts the current state together with the detector type to its neighbors. Two categories of nodes may receive and handle this state, the first category includes the receiver *B* and the second category contains a group of nodes that have recently transmits packets to *A* or *B* denoted as { C_i }. Based on their local knowledge, they can make different transitions. In this example, if *B* receives the *Start* state, it checks its local evidences for the fact that if it has just received many duplicated data packets from *A*. Since recording all detailed acknowledgment information is costly, here the duplicate reception is used as an indication of whether *B* has received and acknowledged the data packets from *A*. If *B* has acknowledged *A* but still receives the same data packet from *A*, it means that *A* successfully sends the packets to *B* but fails to receive the acknowledgment from *B*, otherwise it means that *B* has difficulties in receiving data packets from *A*. The arcs with conditions *B* -/-> *A* and *A* -/-> *B* represent the two situations, respectively. Based on *B*'s local evidences, we can transit the state from *Start* to S_1 or S_2. In this example, as illustrated by the blue arc in Fig. 3, *B* does not receive enough data packets from *A*, so *B* transits the state to S_2 and rebroadcast this state. S_2 can be handled by nodes in { C_i }. If the local information of C_i shows that the data delivery from C_i to *B* is successful, it can be inferred that the poor link quality on link *A* –> *B* leads to the high packet loss (frequent retransmissions). In this case, C_i transits the detector state to an *Accept* state $L_{A->B}$ which indicates a poor link quality on link *A* -> *B*. Otherwise, if C_i can hardly transmit data packets to *B* as well, the state is transited to S_7 as shown by the blue arc. Note that the state S_7 has self-loops on condition C_i -/-> *B*. if the state with self-loops has an output arc labeled with *NUM*, it indicates that this state has a threshold on the maximum number of loops. Take S_7 for example, if more than a certain number of nodes say that they have problems in communicating with *B* (C_i -/-> *B*), we transit the state to the *Accept* state B_c which indicates that there are severe contention at *B* and node *B* is probably congested.

One potential issue in this design lies in the fact that the diagnosis messages can also be lost due to the network failures. For example, as shown in Fig. 2 if node *B* is congested, it may have difficulties in receiving data packets as well as diagnosis message (state of a fault detector) from *A*. However, our fault detectors are not intended for single node, all neighbors which heard the detector can join the diagnosis process as long as it has related evidence. In the above example, though node *B* fails to hear the diagnosis message from *A* due to the congestion, some other nodes { C_i } may forward the diagnosis process and reach the conclusion that *B* is congested.

Another issue is that it is difficult to make different sensor nodes achieve consistent diagnosis results due to the distributive manner of TinyD2. Since different nodes have different

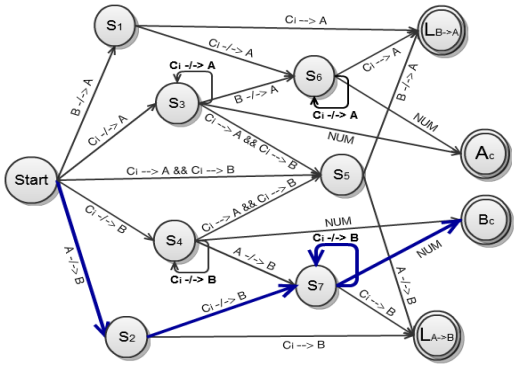

Figure 3 Self-diagnosis with fault detectors

views of the network, they would reach different conclusions for the same fault detector. In this case, sink will tolerate all these results and conduct further analysis by querying more information.

B. *Message Exchanging and Report Handling*

The message exchanged during the diagnosis process include four major parts, the source node ID that creates the fault detector, the detector type, current state of the detector and other supplementary information. In the example of Fig. 3, we add the ID of node *B* as the supplementary information to a diagnosis message.

Upon receiving the state of a new fault detector, the sensor node will check whether it can contribute to this fault diagnosis task. If it has some knowledge, the sensor will transit the state of the corresponding fault detector and propagate the new state, otherwise it simply drop or broadcast the state to other nodes according to the lifetime of this detector. Note that each fault detector has a limitation (similar to TTL) on the number of hops it is delivered.

When the final diagnosis decision is made at some node, it will try to report the decision to sink. If further information is required by the sink, the corresponding sensor nodes will start the active information collection components. If the local region is separated from the integrated network and the sensor node cannot find a route to deliver the diagnosis results to sink, it will periodically broadcast the diagnosis decisions. Through deploying sniffers, we can obtain these diagnosis results. Obviously, more and more diagnosis reports will accumulate at sensor nodes over time and consume the storage space. In this approach, we let sensor node cache each diagnosis report for a specified period and then drop the outdated reports.

C. *Diagnosis Process Triggering*

Frequent diagnosis processes may lead to high transmission and computation cost in a local region. Triggering more fault detectors can improve the fault detection ratio and reduce the detection delay at the cost of higher overhead. To address this issue, we propose a triggering strategy based on the change-point analysis method.

We investigate two categories of evidences for triggering diagnosis. The first category of evidences are the occurrences of special events, such as the sensor node changes its parent, a neighbor has just been removed from the neighbor table of current node, and the like. Once these specified events are detected, we directly trigger a new diagnosis process. The second category of evidences come from the changing of certain parameter values, such as the retransmission frequency, ingress and egress traffic, and the like. We believe that some abnormal symptoms take place only when a certain parameter experiences a significant change in its value, for example, the ingress traffic of a node significantly increases or decreases. In these cases, a new fault detector is created.

The major issue during this process is to determine whether a significant change has taken place in the parameter values. In this work, we form this problem as a change-point detection problem by regarding the sampled values of a parameter during the last period as a time series. Change-points are defined as the abrupt variations in the generative parameters of a time series. By performing a change-point analysis, we can answer the following questions. Did a change occur? When did the change occur? With what confidence did the change occur? There are many existing solutions for change-point detection and analysis. Considering the limited computation and storage resources in sensor nodes, in this work we apply a light-weight approach which combines the cumulative sum charts (CUSUM) and bootstrapping to detect changes.

1) *Cumulative Sum Charts*

We take the traffic as an example to illustrate the diagnosis triggering scheme. In TinyD2, we apply a window-based scheme for caching the latest parameter values. As shown in Fig. 4 (a), assume that the window size is 12 and thus a sensor node stores 12 latest data points of its ingress traffic. At the first step, we calculate the cumulative sum charts of this data sequence. Let $\{ X_i \}$ $i = 1, 2, \ldots, 12$ denote the data points in this stream and $\overline{X}$ be the mean of all values. The cumulative sums are represented as $\{ C_i \}$ $i = 0, 1, \ldots, 12$. Here we define $C_0 = 0$ and then the other cumulative sums are calculated by adding the difference between current value and the mean value to the prior sum.

$$C_i = C_{i-1} + (X_i - \overline{X}) \quad (1)$$

The CUSUM series are illustrated in Fig. 4 (b). In fact, the cumulative sums here are cumulative sums of the differences between values and the mean value. The CUSUM reaches zero at the end. An increase of the CUSUM value indicates that the values in this period are above the overall average value and a descending curve means that values in the corresponding period are below the overall average.

A straight line in the cumulative sum charts indicates that the original values are relatively stable, in contrast, bowed curves are caused by variations in the initial values. The

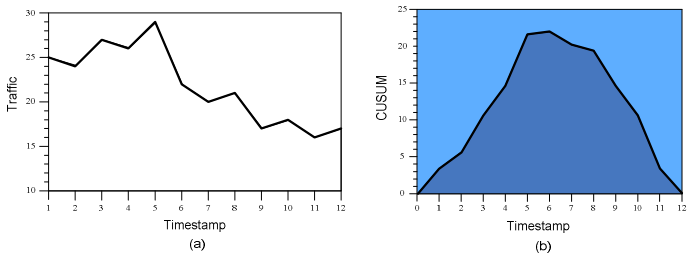

Figure 4 Cumulative sum charts

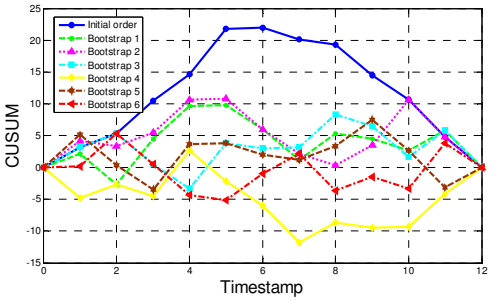

Figure 5 Comparison of the bootstrapping results

CUSUM curve in Fig. 4 (b) turns in direction around C_6 and we can infer that there is an abrupt change. Besides making decision directly according to the CUSUM charts, we also propose to assign a confidence level to our determination by a bootstrap analysis.

2) *Bootstrap strategy*

Before discussing the bootstrap scheme, we firstly introduce an estimator (D_c) of the change which is defined as the difference between maximum value and minimum value in { C_i }.

$$D_c = max(C_i) - min(C_i) \quad (2)$$

In each bootstrap, we randomly reorder the original values and obtain a new data sequence, that is, { X_i^j } j = 1, 2, …m where m denote the number of bootstraps we performed. Then the cumulative sums { C_i^j } as well as the corresponding D_c^j are calculated based on the new sequence { X_i^j }.

Figure 5 shows the cumulative sum curves of the original data sequence and six randomly reordered sequences in bootstraps. The idea behind bootstrap is that randomly reordered data sequences simulate the behavior of CUSUM if no change has occurred. With multiple bootstrap samples we can estimate the distribution of D_c without value changes. In the example of Fig. 5, the bootstraps curves are closer to zero than the CUSUM from values in original order, we can infer that a change must have taken place. We then derive the confidence level by comparing the D_c calculated from values in original order with that from the bootstrap samples.

$$Confidence = \text{number of } (D_c > D_c^j) / m \quad (3)$$

Where D_c is calculated from the original data sequence and D_c^j is derived from a bootstrap, m is the total number of bootstraps performed. If the confidence is above a pre-specified threshold, for example 90%, we decide that there is an apparent change in the parameter values.

3) *Localizing the change-point*

After determining the occurrence of a change, we localize the position of the change with a simple strategy by finding C_i with the largest abstract value. In this example, C_6 has the largest abstract value, so we decide that the value change occurs between X_6 and X_7. The results are shown in Fig. 6. Note that this method can easily be generalized to find multiple changes, however, in this work we only cache and process parameter values in a short period, so we only pick out the major change and trigger a diagnosis process.

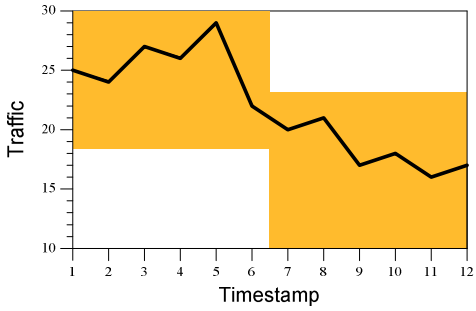

Figure 6 Change-point analysis results

IV. TINYD2 IMPLEMENTATION

We have implemented the TinyD2 on the Telosb mote with a MSP430 processor and CC2420 transceiver. On this hardware platform, the program memory is 48K bytes and the RAM is 10K bytes. We apply the TinyOS 2.1 as our software development platform and the application and TinyD2 diagnosis functionalities are implemented as different components.

In the diagram of Fig. 7, components in the gray region belong to TinyD2. The *Information Collection* component collects and caches system information from other components. The *Diagnosis Triggering* component dynamically examines the system information and starts a fault diagnosis process when abnormal situations detected. The *Local Diagnosis Controlling* component transits the state of fault detectors according to the information from the *Information Collection* component and thus to advance the diagnosis process. The *Message Exchanging* component takes responsible for exchanging intermediate detector states with other sensor nodes. The *Report Processing* component is in charge of dealing with the final diagnosis report. When a final diagnosis decision is made at current node, the *Report Processing* component firstly tries to deliver the diagnosis results to sink. If the route between current node and sink is unavailable, the *Report Processing* component will broadcast the reports to its neighbors. For some fault types, sink needs to retrieve further information from sensor nodes, in this case the *Report Processing* component will handle these queries. The above 5 components provide

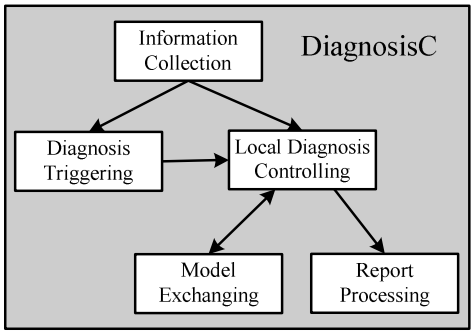

Figure 7 Software diagram of TinyD2

interfaces of their corresponding functions and *DiagnosisC* component uses these interfaces and provides the integrated interface *Diagnosis*, the ROM usage increases by 7.3K bytes.

V. EXPERIMENTS

In the section, we describe performance evaluation results of our TinyD2 approach. We conduct experiments to test the accuracy and energy efficiency of TinyD2 with manually injected faults in a 100 nodes test bed.

We test the performance of TinyD2 in a test bed with 100 sensor nodes. These sensor nodes are powered by the USB hardware interface. We implement TinyD2 on sensor motes and evaluate its performance and overhead by manually injecting varying types of errors. We employ two metrics for measuring the diagnosis performance of TinyD2, the *detection ratio* and the *false alarm ratio*. The *detection ratio* is defined as the ratio between the number of faults detected and the number of all faults. Higher *detection ratio* can help the network managers recover more failures. The *false alarm ratio* is the ratio between false alarms and all diagnosis reports. A low *false alarm ratio* indicates that the diagnosis tool has a high accuracy.

In our experiments, we compare the performance of TinyD2 with PAD, one of the recent diagnosis approaches for sensor networks. PAD proposes the concept of passive diagnosis which collects the network information by marking routine data packets and deduces the root cause with a probabilistic inference model. PAD is efficient both in diagnosis performance and the energy consumption. During these tests, we manually inject three types of failures, the node crash, link failures and the bad routes. We also vary the network size from 20 to 100.

Figure 8 shows the results of node crash detection. According to Fig. 8 (a), TinyD2 successfully finds more than 90% of the crashed nodes and the *detection ratio* keeps stable as the network size increases. PAD achieves a more than 90% *detection ratio* in a small network, however, when network size increases its detection ratio decreases to around 80% (in the network with 100 nodes). As shown in Fig. 8 (b), the *false alarm ratio* of TinyD2 is around 10% over varying network sizes and PAD has more false alarms in larger networks.

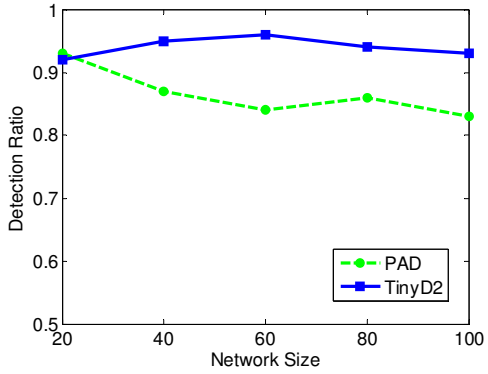

Figure 8 (a) The detection ratio for node crash V.S. Network size

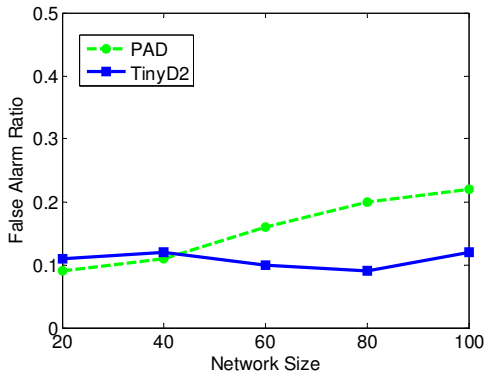

Figure 8 (b) The false alarm ratio for node crash V.S. Network size

This is because PAD relies on the probabilistic inference model to deduce the root cause, when the network size increases, the model turns to be more complicated and thus incurs more wrong results. Instead, TinyD2 encourages all sensor nodes to join the diagnosis process and makes decisions based on the local evidences of the sensor nodes and thus can provide stable diagnosis results in spite of the overall network size.

Figure 9 (a) and (b) illustrate the *detection ratio* and *false alarm ratio* of the two approaches on diagnosing the link failures over different network sizes. We observe similar trends in these results. TinyD2 achieves a *higher detection ratio* and lower *false alarm ratio* than PAD and the performance of TinyD2 is more stable. In Fig. 10 (a) and (b), we show the results of diagnosing the bad routes failures. As the probabilistic inference model in PAD doesn't output the bad routes failure directly, so we only present the results of TinyD2. According to the results in Fig. 8 - 10, we can conclude that TinyD2 achieves similar performances on detecting all these three types of failures.

We then evaluate the overhead of TinyD2. We firstly compare the traffic overhead of TinyD2 with that of PAD. We apply the ratio between diagnosis traffic and the overall network traffic to quantify the overhead. Figure 11 shows the CDF of

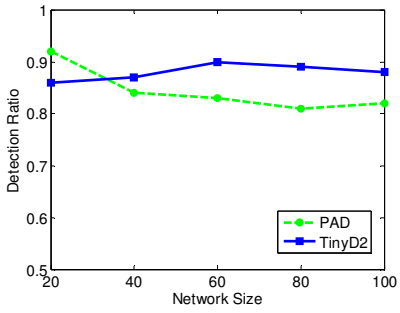

Figure 9 (a) The detection ratio for link failures V.S. Network size

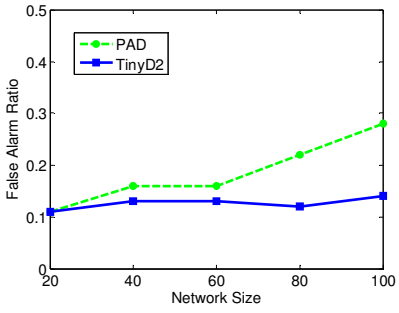

Figure 9 (b) The false alarm ratio for link failures V.S. Network size

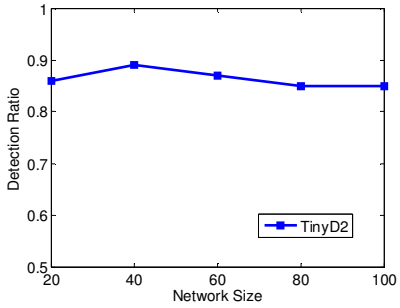

Figure 10 (a) The detection ratio for bad routes V.S. Network size

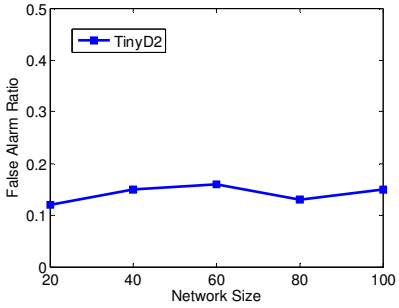

Figure 10 (b) The false alarm ratio for bad routes V.S. Network size

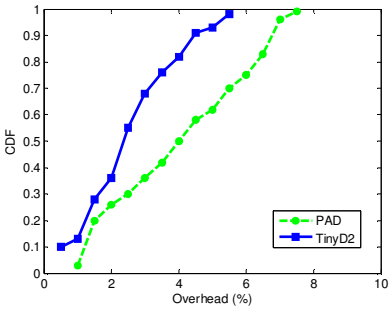

Figure 11 Diagnosis overhead (TinyD2 V.S. PAD)

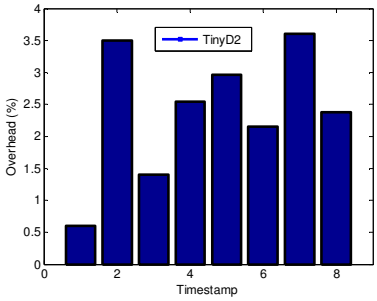

Figure 12 Overhead over time

the two approaches' traffic overhead. The curve of TinyD2 is on the left of the curve of PAD which indicates that the cumulative overhead of TinyD2 is less than PAD. For example, according to Fig. 11, at 80% of the cases the TinyD2 only uses less than 3.5% of overall network traffic while PAD uses less than 6%. We also evaluate the overhead of TinyD2 over time. As shown in Fig. 12, we find that the overhead of TinyD2 exhibits high variations over time. In some timestamps, the diagnosis overhead is less than 1% and in some other timestamps the overhead can reach to nearly 4%. This is because the TinyD2 is an on-demand approach which only works when network failures occur.

VI. Conclusion and Future Works

In this work, we propose a novel self-diagnosis approach TinyD2 for large scale sensor networks. Existing diagnosis approaches are mainly sink-based which suffer from the high communication overhead and the incomplete diagnosis information. Instead, TinyD2 employs the self-diagnosis concept which encourages all sensor nodes to join the diagnosis process. To address the issue of single nodes having insufficient information for determining the root-causes for many failures, TinyD2 presents a series of novel fault detectors through which multiple nodes can cooperate with each other in a diag-

nosis task. The fault detectors encode the diagnosis process to state transitions. Through comprehensive experiments in indoor testbed, we compare this design with existing approaches. Currently our fault detectors are deterministic, and in future works, we plan to extend our model to cope with nondeterministic cases.

ACKNOWLEDGMENT

This work is supported in part by National Basic Research Program of China (973 Program) under Grants 2010CB328000 and No. 2011CB302705, the National Natural Science Foundation of China (Grant No. 61073168) and NSFC/RGC Joint Research Scheme N HKUST602/08.

REFERENCE

[1] "HP Openview."

[2] "IBM Tivoli."

[3] "Microsoft Operations Manager."

[4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, "Towards Highly Reliable Enterprise Network Services Via Inference of Multi-Level Dependencies," In Proc. of ACM SIGCOMM, 2007.

[5] 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 Proc. of ACM SenSys, 2008.

[6] B.-r. Chen, G. Peterson, G. Mainland, and M. Welsh, "LiveNet: Using Passive Monitoring to Reconstruct Sensor Network Dynamics," In Proc. of IEEE/ACM DCOSS, 2008.

[7] Z. Chen and K. G. Shin, "Post-Deployment Performance Debugging in Wireless Sensor Networks," In Proc. of IEEE RTSS, 2009.

[8] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "EmStar: A Software Environment for Developing and Deploying Wireless Sensor Networks," In Proc. of USENIX Annual Technical Conference, 2004.

[9] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection Tree Protocol," In Proc. of ACM SenSys, 2009.

[10] S. Guo, Z. Zhong, and T. He, "FIND: Faulty Node Detection for Wireless Sensor Networks," In Proc. of ACM SenSys, 2009.

[11] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh, "Energy-Efficient Surveillance System Using Wireless Sensor Networks," In Proc. of ACM MobiSys, 2004.

[12] S. Kandula, D. Katabi, and J.-P. Vasseur, "Shrink: A Tool for Failure Diagnosis in IP Networks," In Proc. of MineNet Workshop at ACM SIGCOMM, 2005.

[13] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, "Detailed Diagnosis in Enterprise Networks," In Proc. of ACM SIGCOMM, 2009.

[14] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, "Dustminer: Troubleshooting Interactive Complexity Bugs in Sensor Networks," In Proc. of ACM SenSys, 2008.

[15] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, "IP Fault Localization Via Risk Modeling," In Proc. of USENIX NSDI, 2005.

[16] M. Li and Y. Liu, "Underground Structure Monitoring with Wireless Sensor Networks," In Proc. of IEEE/ACM IPSN, 2007.

[17] P. Li and J. Regehr, "T-Check: Bug Finding for Sensor Networks," In Proc. of IEEE/ACM IPSN, 2010.

[18] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong, "Passive Diagnosis for Wireless Sensor Networks," In Proc. of ACM SenSys, 2008.

[19] A. Mahimkar, Z. Ge, A. Shaikh, J. Wang, J. Yates, Y. Zhang, and Q. Zhao, "Towards Automated Performance Diagnosis in a Large IPTV Network," In Proc. of ACM SIGCOMM, 2009.

[20] L. Mo, Y. He, Y. Liu, J. Zhao, S. Tang, and X. Li, "Canopy Closure Estimates with GreenOrbs: Sustainable Sensing in the Forest," In Proc. of ACM SenSys, 2009.

[21] R. Rajagopal, X. Nguyen, S. C. Ergen, and P. Varaiya, "Distributed Online Simultaneous Fault Detection for Multiple Sensors," In Proc. of IEEE/ACM IPSN, 2008.

[22] N. Ramanathan, K. Chang, L. Girod, R. Kapur, E. Kohler, and D. Estrin, "Sympathy for the Sensor Network Debugger," In Proc. of ACM SenSys, 2005.

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

[24] G. Tolle and D. Culler, "Design of an Application-Cooperative Management System for Wireless Sensor Networks," In Proc. of IEEE EWSN, 2005.

[25] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis, "Visibility: A New Metric For Protocol Design," In Proc. of ACM SenSys, 2007.

[26] A. Woo, T. Tong, and D. Culler, "Taming the Underlying Challenges of Reliable Multihop Routing in Sensor Networks," In Proc. of ACM SenSys, 2003.

[27] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A Wireless Sensor Network for Structural Monitoring," In Proc. of ACM SenSys, 2004.

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

[29] Z. Yang, M. Li, and Y. Liu, "Sea Depth Measurement with Restricted Floating Sensors," In Proc. of IEEE RTSS, 2007.

[30] J. Zhao, R. Govindan, and D. Estrin, "Residual Energy Scan for Monitoring Sensor Networks," In Proc. of IEEE WCNC, 2002.