SNIF: Sensor Network Inspection Framework

Matthias Ringwald, Kay Römer
Institute for Pervasive Computing
ETH Zurich, Switzerland
Email: {mringwal,roemer}@inf.ethz.ch

Andrea Vitaletti
Department of Informatics
University of Rome "La Sapienza", Italy
Email: andrea.vitaletti@dis.uniroma1.it

***Abstract*— Recent experience with the deployment of sensor networks demonstrates that it is far from trivial to setup a working larger-scale sensor network in the field. Even though simulations and experiments with lab testbeds confirmed a working system, subtle real-world influences lead to frequent failures in the field. Identifying and fixing these problems in the field is currently a difficult and cumbersome task due to the lack of appropriate concepts and tools. In this paper we address this issue by, firstly, classifying common problems that have been encountered during deployment. We then show that many of these problems can be detected by overhearing and analyzing sensor network traffic without need for an instrumentation of sensor nodes. Based on this observation, we develop a tool to inspect a deployed sensor network, consisting of a distributed network sniffer and a data-stream-based framework for online traffic analysis. We demonstrate and evaluate how this tool can be used to debug a typical data gathering application.** [1]

I. Introduction

Deploying large-scale sensor networks in real-world settings is a challenging issue [5], [11], [17], [18], [19], [20], [26], [28], [31]. After putting sensor nodes in place in the field, the network usually does not perform as expected or refuses to work altogether. Then, the behavior of the network and individual nodes must be inspected and understood to identify and fix the problem. Since this has to be performed in situ in the field, deployment is currently a costly and cumbersome task and may be considered as an important obstacle towards the wide-spread adoption of sensor networks for real-world applications. Even though tools for simulation and emulation as well as lab testbeds are available today, structured approaches and tools to support deployment in the field are largely missing. The reasons that make deployment of sensor networks difficult can be summarized in the following way.

What works in the lab fails in the field w.h.p.: Sensor networks are deeply embedded into the physical world. Therefore, their function heavily depends on the physical environment, which can only very rudimentarily be captured in simulators, emulators, or lab testbeds. Not only does the physical world control the output of sensors, but it influences the behavior of wireless links and sensor nodes in significant ways. In addition, lab testbeds are often limited in scale due to the required cabling infrastructure. Hence, even though simulations or experiments with lab testbeds ran successfully, the larger scale of real-world deployments and the impact of the real world are likely to break the sensor network. Therefore, deployment typically involves significant work in the field.

Monitoring the network state is difficult: Once things go wrong, the sensor network needs to be inspected in the field in order to understand the cause of the problem. Current practice requires instrumentation of sensor nodes with monitoring software and monitoring traffic is sent in-band with the sensor network traffic to the sink (e.g., [20], [22], [30]). This approach has three major drawbacks. Firstly, problems in the sensor network (e.g., partitions, message loss) also affect the monitoring mechanism, thus reducing the desired benefit. Secondly, since the nature of problems is often unknown prior to deployment, sensor nodes may need code updates in the field in order to capture data relevant to find a problem. Thirdly, sensor network resources (cpu cycles, memory, network bandwidth) are used for inspection. Due to the scarcity of these resources, it is highly desirable to remove the instrumentation as soon as the network works as expected. This, however, may lead to significant *probe effects* [10], where the behavior of a sensor network is altered by adding/removing instrumentation mechanisms. In the Sympathy debugging system [20], for example, up to 30% of the network bandwidth is used for monitoring traffic.

The aim of this paper is two-fold. Firstly, we contribute to a more systematic understanding and treatment of deployment issues. For this purpose, we studied the existing literature on deployment experience and present a classification of common problems encountered during deployment of sensor networks in Sect. II along with a set of indicators in Sect. III that – when observed – hint the existence of a specific problem.

Secondly, we present and evaluate an extensible

[1]The work presented in this paper was partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322.

framework for inspection of deployed sensor networks in the field that does not require an instrumentation of sensor nodes. Our approach uses a so-called *deployment support network* (DSN) [4]: a wireless network that is temporarily installed alongside the actual sensor network during the deployment process. Each DSN node provides two different radio front-ends. The first radio is used to overhear the traffic of the sensor network, while the second radio is used to form a robust and high-bandwidth ad hoc network among the DSN nodes to deliver overheard packets to an observer. The overheard sensor network traffic is then fed to a data-stream-based framework in order to find indicators for the existence of problems. The framework provides an extensible set of parameterizable data stream operators for online analysis of network traffic. In Section VI we demonstrate how this framework can be used to debug a typical data gathering application. An evaluation of this approach in Sect. VII shows that our approach can detect problems accurately and timely and can deal well with incomplete information.

II. Problems

This section contains a classification of the problems typically found during deployment according to our own experience and as reported in the literature. Here, a problem is essentially defined as a behavior of a set of nodes that is not compliant with the specification.

We classify problems according to the number of nodes involved into four classes: *node problems* that involve only a single node, *link problems* that involve two neighboring nodes and the wireless link between them, *path problems* that involve three or more nodes and a multi-hop path formed by them, and *global problems* that are properties of the network as a whole.

A. Node Problems

A common node problem is *node death* due to energy depletion either caused by "normal" battery discharge or due to short circuits. In [27], a low-resistance path between the power supply terminals was created by water permeating a capacitive humidity sensor, resulting in early battery depletion and abnormally small or large sensor readings.

Low batteries often do not cause a fail-stop behavior of the sensor nodes. Rather, nodes may show Byzantine behavior at certain low battery voltages. As reported in [31], for example, *strange sensor readings* (temperature values $> 100°$C) have been observed at low battery voltage.

Software bugs often result in *node reboots*, for example, due to failure to restart the watchdog timer of the micro controller [15]. We also observed software bugs resulting in hanging or killed threads, such that only part of the sensor node software continued to operate.

Many sensor applications are periodic (e.g., sample, receive, send, sleep) with a fixed cycle length. Relative drift of the phases of these cycles among nodes may result in failure to deliver data as nodes are not ready to receive when others are sending, or increased congestion as many nodes send concurrently. In [17], excessive *phase skew* has been observed (about two orders of magnitude larger than the drift of the oscillator).

B. Link Problems

Field experiments (e.g., [8], [33]) demonstrated a very high variability of link quality both across time and space resulting in temporary link failures and variable amounts of *message loss*. *Network congestion* caused by collisions is another source of message loss. In [26], for example, a median message loss of 30% is reported for a single-hop network. Excessive levels of congestion have been caused by accidental synchronization of transmissions by multiple senders, for example, due to inappropriate design of the MAC layer [21] or by repeated network floods [15]. If message loss is compensated for by retransmissions, a *high latency* may be observed until a message eventually arrives at the destination. Most sensor network protocols require each node in the sensor network to discover and maintain a set of network neighbors (often implemented by broadcasting HELLO messages containing the sender address). A node with *no neighbors* presents a problem as it is isolated and cannot communicate. Also, *neighbor oscillation* is problematic [21], where a node experiences frequent changes of its set of neighbors.

A common issue in wireless communication are *asymmetric links*, where communication between a pair of nodes is only possible in one direction. In a field experiment [8] between 5-15% of all links have been observed to be asymmetric, with lower transmission power and longer node distance resulting in more asymmetric links. If not properly considered, asymmetric links may result in fake neighbors (received HELLO from a neighbor but cannot send any data to it) and broken data communication (can send data to neighbor, but cannot receive acknowledgements).

Another issue is the length of a link. Even though two nodes are very close together, they may not be able to establish a link (*missing short links*). On the other hand, two nodes that are very far away from each other (well beyond the nominal communication range of a node), may be able to communicate (*unexpected long links*). Experiments in [8] show that at low transmit power about 1% of all links are twice as long as the nominal communication range. These link characteristics make node placement highly non-trivial.

C. Path Problems

Many sensor network applications rely on the ability to relay information across multiple nodes along a multi-hop path. In particular, most sensor applications include one or more sink nodes that disseminate queries or other tasking information to sensor nodes and sensor nodes deliver results back to the sink. Here, it is important that a path exists from a sink to each sensor node, and from each sensor node to a sink. Note that information may be changed as it is traversing such a path, for example due to data aggregation. Two common problems in such applications are hence *bad path to sink* and *bad path to node*. In [15], for example, *selfish nodes* have been observed that did not forward received traffic, but succeeded in sending locally generated messages.

Since a path consists of a sequence of links, the former inherits many of the possible problems from the latter such as *asymmetric paths*, *high latency*, *path oscillations*, and *high message loss*. In [26], for example, a total message loss of about 58% was observed across a multi-hop network.

Finally, *routing loops* are a common problem, since frequent node and communication failures can easily lead to inconsistent paths if the software isn't properly prepared to deal with these cases. Directed Diffusion [14], for example, uses a data cache to suppress previously seen data packets to prevent loops from taking place. If a node reboots, the data cache is deleted and loops may be created [23].

D. Global Problems

In addition to the above problems which can be attributed to a certain subset of nodes, there are also some problems which are global properties of a network. Several of these are failures to meet certain application-defined quality-of-service properties. These include *low data yield*, *high reporting latency*, and *short network lifetime* [29].

Low data yield means that the network delivers an insufficient amount of information (e.g., incomplete sensor time series). In [31], for example, a total data yield of only about 20-30% is reported. This problem is related to message loss as discussed above, but may be caused by other problems such as a node crashing before buffered information could be forwarded, buffer overflows, etc. One specific reason for a low data yield is a *partitioned network*, where a set of nodes is not connected to the sink.

Reporting latency refers to the amount of time that elapses between the occurrence of a physical event and that event being reported by the sensor network to the observer. This is obviously related to the path latency, but as a report often involves the output of many sensor nodes, the reporting latency results from a complex interaction within a large portion of the network.

The lifetime of a sensor network typically ends when the network fails to cover a given physical space sufficiently with live nodes that are able to report observations to the observer. The network lifetime is obviously related to the lifetime of individual nodes, but includes also other aspects. For example, the death of certain nodes may partition the network such that even though coverage is still provided, nodes can no longer report data to the observer.

III. Indicators

An indicator is an observable behavior of a sensor network that hints (in the sense of a heuristic) the existence of one of the problems discussed in the previous section. In the context of our work we are particularly interested in indicators that can be observed purely by overhearing the traffic of the sensor network as this does not require any instrumentation of the sensor nodes. We call such indicators *passive*. The structure of this section mirrors that of the previous section, discussing passive indicators for the problems outlined there. In fact, passive indicators heavily depend on the protocols used in the sensor network. However, as we will see below there are a number of protocol elements that are commonly found in sensor network applications that can be exploited. For example, many protocols exchange regular beacon messages, all packets need to contain the per-hop destination MAC address, some packets also contain the per-hop source MAC address to identify the sender of the message, and some packets do contain a monotonically increasing sequence number (cf. Sect. VI-A).

A. Node Problems

Node death: Many commonly used MAC and routing protocols (e.g., [33], [12]) require every node to transmit a beacon message at regular intervals, in particular for the purpose of synchronization and neighbor management. Failure to transmit any such message for a certain amount of time (typically a multiple of the inter-beacon time) is an indicator for node death. Also, node death can be assumed if a node is no longer considered a neighbor by any other node (cf. Sect. III-B).

Node reboot: When a node reboots, its sequence number counter will be reset to an initial value (typically zero). Hence, the sequence number contained in messages sent by the node will jump to a smaller value after a reboot w.h.p., which can serve as an indicator for reboot. Note that a reboot cannot be detected this way when the node crashes just before the sequence number counter would wrap around to its initial value. However, a simple fix would be to set the sequence counter to

some value other than the initial value at wrap-around, such that a wrap-around could be distinguished from a reboot.

Phase skew: Again we can exploit the existence of beacon messages that are sent at regular time intervals. Change of the time between two beacons over time indicates phase drift. Averaging over multiple beacon intervals can help eliminate variable delays introduced by, e.g., medium access.

Strange sensor readings: These can only be passively observed when application messages contain raw sensor readings.

B. Link Problems

Discovering neighbors: Unfortunately, the neighbor table of a node cannot be passively observed directly. However, there are two ways to learn about the neighbors of a node. Firstly, by overhearing the destination ids of messages a node sends we can learn a subset of the neighbors. The second approach exploits link advertisements sent by nodes to estimate link quality. Since links are often asymmetric in sensor networks, link quality estimation must consider both directions of a link. Since a sensor node can only measure the quality of one direction of a link directly (e.g., by means of the fraction of beacon messages being received), nodes broadcast link advertisement messages containing the addresses and quality of the incoming links from their neighbors (cf. Sect. VI-A). These messages can be passively observed to obtain information about the neighbors of a node and the quality of the associated link.

Knowing the neighbors of a node, we can detect neighbor oscillation and nodes with no neighbors. If the locations of nodes are known, we can also discover missing short links and unexpected long links.

Message loss: Again, it is not directly possible to decide whether or not a node has received a message by means of passive observation. However, in many situations reception of a message by a node does trigger the transmission on another message by that node (e.g., acknowledgment, forwarding a message to the next hop). If such a property exists, failure to overhear the second message within a certain amount of time after the first message has been overheard is an indicator for message loss. Note that with this approach, it is not possible to tell apart message loss from selfish nodes.

Latency: To estimate the latency of a link, the same approach as for detecting message loss is used. The time elapsed between overhearing the causal and the consequential message gives an estimate of the latency, which includes processing delays in the node.

Congestion: The level of link congestion perceived by a sensor node cannot be passively observed. The level of congestion experienced by a deployment support node overhearing the traffic that is being addressed to this sensor node can be used as a rough approximation. Alternatively, message loss and latency as discussed above could be used as indicators for the level of congestion.

C. Path Problems

Discovering paths: In order to discover the path between two sensor nodes (e.g., from node to sink and from sink to node), we would need access to the routing tables. As for the neighbor table, this is not directly possible with passive observation. As for neighbor discovery, there are two possible ways around. Firstly, we can reconstruct a path by tracking a message as it travels from source to destination. Using the receiver ids of the overheard messages, we can reconstruct nodes on the path. However, for this we must be able to decide whether two overheard messages belong to the same path. This is trivial if the message payload is relayed unmodified along the path. However, it is highly non-trivial if data is aggregated along the path. In such cases one may resort to a temporal correlation among the overheard messages. Secondly, we can overhear routing messages if there exist any. While these messages typically indicate that a node has established a route, it is often impossible to reconstruct the route. To construct a spanning tree, for example, it is sufficient that nodes broadcast messages containing their address and distance to the sink, but not their parent in the tree (cf. Sect. VI-A). The latter would be necessary to reconstruct the spanning tree from overheard traffic.

If paths can be discovered, we can also easily detect path oscillations and find missing paths from nodes to sink and vice versa. Using similar techniques as for links, we can estimate message loss and latency along a path.

Loops: Duplicate messages being sent to the same node are an indicator for loops. Care must be taken to not confuse retransmissions and loops. Alternatively, the discovered paths can be directly examined for loops.

D. Global Problems

As discussed in Section II, global problems such as low data yield, high reporting latency, or insufficient network lifetime are typically due to a combination of different node, link, and path problems. Hence, the indicators for the latter problems discussed above can be considered as indicators for these global problems.

Partitions: Knowledge of the neighbors of each node as discussed above allows to reconstruct the network topology and the detection of network partitions.

E. Discussion

As we have shown throughout this section, for many common problems that occur during deployment of

sensor networks, passive indicators exist that allow to infer the existence of a problem from overheard network traffic. However, in some situations we had to make assumptions about the underlying sensor network protocols that may not hold for all applications (e.g., indicators for message loss). For other problems, passive indicators provide only an approximate view of the ground truth (e.g., indicators for network congestions).

In such situations, we can resort to *semi-active observation* or *semi-passive observation.* With *semi-active observation*, the sensor nodes are instrumented with code to emit messages containing additional information about the state of the sensor node (e.g., level of congestion, battery voltage, etc.). These messages are overheard by the DSN and ignored by other sensor nodes. While this approach requires instrumentation of sensor nodes and transmission of additional messages, these messages do not have to be routed through the sensor network. Semi-active observation is also supported by SNIF.

With *semi-passive observation*, message can also be injected into the sensor network. This approach may for example be used to probe for the liveness and reactivity of a sensor node, or to replay pre-recorded sequences of messages to check the behavior of the sensor network regarding global properties such as reporting latency. We plan to support semi-passive observation with future versions of SNIF.

IV. Passive Observation

This section presents the hardware and software infrastructure needed to overhear the traffic of a sensor network. Here, a sensor network is essentially a multi-hop ad hoc wireless network of sensor nodes including one or more sink nodes that act as gateways to the “user”. To overhear the traffic of such a sensor network, we essentially need a set of distributed radios to listen to the traffic: the *deployment support network*. Each of these radios must implement the receiver part of the sensor network protocol stack, namely read-only *physical layer* and *media access*, as well as a *packet decoder* to extract the contents of overheard packets.

A. Deployment Support Network (DSN)

A deployment support network is a wireless ad hoc network of deployment support nodes. Each deployment support node provides two radios. The first radio (DSN radio) is used to form a wireless network among the deployment support nodes, while the second radio (WSN radio) is used to overhear the traffic of the sensor network. Both radios should be free of interference (e.g., by operating in different frequency bands). Also, the DSN radio should support the formation of a robust multi-hop network with high bandwidth (compared to the WSN). In particular, the deployment support network should have negligible message loss. Messages overheard by a DSN node are (pre)processed and routed to the DSN sink, where they are analyzed to detect indicators (cf. Sect. V).

The nodes of a deployment support network are internally time-synchronized to time-stamp overheard packets. Hence, the synchronization accuracy should be less than the time needed to receive a packet with the WSN radio (i.e., in the order of milli seconds for a typical 19.2 kbps WSN radio).

The DSN is installed alongside the actual sensor network and may be removed again as soon as the sensor network works as expected. Thus, the lifetime of the DSN is typically much shorter than the lifetime of the sensor network and energy efficiency is not that much of an issue.

Our current implementation of a DSN is based on the BTnode [35], which provides two radio front-ends: a Zeevo ZV 4002 Bluetooth 1.2 radio which is used as the DSN radio, and a Chipcon CC 1000 which is used as the WSN radio. The CC 1000 is, for example, also used on the popular MICA2 motes [34] sold by Crossbow.

Using a scatternet formation algorithm, the DSN nodes form a robust Bluetooth scatternet. A laptop equipped with Bluetooth acts as DSN sink that forms the root of an overlay tree spanning the whole DSN. The sink can send data to DSN nodes down this tree, DSN nodes report data (i.e., overheard packets) up the tree to the sink. The implementation of the Bluetooth scatternet and data routing are described elsewhere [4]. The focus of this paper is on the use of this deployment support network for passive observation of sensor networks, not on networking aspects of the DSN.

Note that the above is only one example for a DSN. Many other implementations are possible. For example, one might simply implement a DSN node by connecting a sensor node via USB to a PDA equipped with 802.11. The WSN node is used to overhear the traffic of the sensor network and 802.11 is used to form a robust, high-bandwidth, and long-range deployment support network.

B. Physical Layer and Medium Access

DSN nodes need a receive-only implementation of the physical (PHY) and MAC layers in order to overhear sensor network traffic. Due to the lack of a standard protocol stack, many variants of PHY and MAC are in use in sensor networks. Hence, we need a flexible implementation that can be easily configured for the sensor network under inspection.

Regarding PHY, important configuration parameters are the carrier frequency, baud rate, and checksumming details. In particular, we require that the sensor network uses a single frequency for communication (which is the

```
 1 // Physical+MAC Layers
 2 cc.freq = 868000000;
 3 cc.baud=19200;
 4 cc.sop = 0x55aa;
 5 cc.crc = 0xF1F1;
 6 // Encoding: endianness + alignment
 7 encoding.endianness = "little";
 8 encoding.alignment = 1;
 9 // Used Types
10 typedef uint16_t  mote_id_t;
11 typedef uint8_t   quality_t;
12 struct link_quality_t {
13     mote_id_t id;
14     quality_t quality;
15 };
16 // Constants
17 const int LINKESTADV = 2;
18 // Packet types
19 default.packet = "TOS_Msg";
20 struct TOS_Msg {
21     uint16_t addr;
22     uint8_t type;
23     uint8_t group;
24     uint8_t length;
25     // variable size of packet payload
26     int8_t data[length];
27     uint16_t crc;
28 };
29 struct LinkAdvertisement :
30   TOS_Msg.data ( type == LINKESTADV ) {
31     mote_id_t id;
32     // variable nr of link quality entries
33     struct link_quality_t links[];
34 };
```

Fig. 1. A SNIF configuration file.

case with current implementations) such that as single-channel radio is sufficient to overhear WSN traffic.

Regarding MAC, every DSN node has its WSN radio turned to receive mode all the time (note that DSN lifetime is relatively short). In the arriving stream of bits, the MAC layer implementation looks for a start-of-packet (SOP) delimiter that must be sent before each packet in order to synchronize sender and receiver. Once an SOP has been found, payload data and a CRC follow. In case of variable-sized packets, a length indicator must be contained in the payload data, see Sect. IV-C for details. This way, we can receive packets independent of the actual MAC layer used.

Figure 1 shows an excerpt of a sample configuration file for a TinyOS application running on the MICA2 mote. The first five lines set the carrier frequency of the WSN radio to 868.000 Mhz and a data rate of 19200 baud, and instruct the packet sniffer to check for a start-of-packet sequence of 0x55aa. The used CRC polynomial is 0xF1F1.

C. Packet Decoder

Again, since no standard protocols exist for sensor networks, we need a flexible mechanism to decode overheard packets. Since most programming environments for sensor nodes are based on the C programming language or a dialect of it (e.g., nesC for TinyOS), it is common to specify message contents as a (nested) C struct in the source code of the sensor network application. The packet decoder uses an annotated version of such C structs as a description of the packet contents. This way, the user can copy and paste packet descriptions from the source code.

The configuration consists of some global parameters (such as byte order and alignment), type definitions, and one or more C structs. One of these structs is indicated as the default packet layout. Note that such a struct can contain nested other structs, effectively implementing a discriminated union.

Consider Fig. 1 for an example, which describes link advertisement packets used by the Multihop routing service implemented in ESS [12]. In line 7-8, the byte order is defined as little endian and the alignment set to be 1. Lines 10-15 contain a number of type definitions. Line 19 defines the struct `TOS_Msg` as the default packet layout. The contents of this structure (i.e., a TinyOS message) are given in lines 20-28. The LinkAdvertisement PDU used by ESS, whose contents are specified in lines 31-33, is encapsulated in the field `TOS_Msg.data`, but only if the `TOS_Msg.type` is equal to `LINKESTADV`. Lines 29-30 specify the contents of `TOS_Msg.data` depending on the value of the field `TOS_Msg.type`. Note that an encapsulated structure can itself also contain another encapsulated structure.

Our description language allows to specify variable sized arrays. If an element of a message is used as array size (e.g., `TOS_Msg.length`), then the value of this field in the overheard message denotes the number of elements in the array to decode. If an array without a size is given (e.g., `LinkAdvertisement.links`), the size of the array is inferred from the total packet size.

At startup of SNIF, the configuration file is parsed and the default packet type is investigated. If the default packet type is of fixed size, the packet size is computed. Otherwise, size and position of the packet length indicator (e.g., `TOS_Msg.length` in the example) is computed. This information, along with the parameters for the physical layer are then broadcast to all DSN nodes, allowing them to correctly receive WSN traffic. All overheard WSN packets are then annotated with reception time and reception quality (i.e., signal strength) and routed to the SNIF sink. The SNIF framework described in Sect. V executing on the DSN sink is then able to decode the packets and access the value of any message field by its symbolic name (e.g., `TOS_Msg.addr`).

D. Deploying a DSN

As mentioned before, a deployment support network needs to be installed alongside the observed sensor network. One might argue that the deployment of the DSN may be as difficult and error-prone as deploying

the sensor network itself. However, the DSN is intended as a tool that is reused unmodified for many deployments and can thus be expected to be largely free of bugs. The inspected sensor network, in contrast, contains newly developed WSN protocols and application code, which are likely to contain errors. Also, since energy constraints are relaxed in the DSN due to its shorter lifetime, we can rely on more robust networking technologies (such as Bluetooth).

Nonetheless, deploying a DSN is a non-trivial issue as we need to ensure *connectivity* and *coverage*. Here, connectivity means that the DSN is not partitioned. A DSN is said to cover a set of sensor nodes if messages transmitted by the latter can be received by the DSN (except for transient message loss). Which portion of a WSN has to be covered depends on the nature of the problems the DSN should be able to detect (cf. Sect. III).

To support the user with the deployment of the DSN, each DSN node provides indicators for coverage and connectivity by means of flashing LEDs. The connectivity indicator is activated as soon as the DSN node can establish a connection to another, already deployed DSN node. The coverage indicator flashes whenever a message is received from the sensor network with a given minimal average signal strength. If beacon messages are sent at regular intervals, the flashing frequency is proportional to the number of sensor nodes that can be overheard with sufficient quality. More detailed information (i.e., IDs of currently covered sensor nodes and observation quality) are displayed at the DSN sink.

E. Discussion

Using a DSN, we are able to passively observe a deployed sensor network without any instrumentation of the sensor network. The resulting key advantages are that WSN software does not have to be touched, inspection does not use any resources in the sensor network, and probe effects are avoided (or minimized when using semi-active observation).

There are also some potential drawbacks involved with using a DSN, however. Firstly, apart from the actual sensor network, additional hardware is required (which may be reused for different deployment projects). Also, the DSN has to be installed and removed. We believe, however, that these overheads are compensated for by saving time and effort during deployment. In this context, it should be noted that different sensor node platforms use different radios, which the DSN has to support. Here, modular platforms with "pluggable" radios would be helpful. Secondly, using the passive observation approach requires a deep understanding of the semantics of the protocols used in the sensor network. Often, it may seem easier to use semi-active observation, which, however requires understanding and modifying the software executing on the sensor nodes.

V. Detection Framework

Using a deployment support network, we can overhear the traffic of a sensor network. To identify problems in the sensor network, the traffic traces need to be analyzed to detect indicators for possible problems. As we want to identify problems as soon as possible after their occurrence, we need to perform an *online* analysis.

While we have identified a set of common problems in Sect. II and suggested passive indicators for them in Sect. III, we want to emphasize that these are not exhaustive. Rather, novel problems may occur or different indicators for known problems may be needed. To support this open problem space, we need an extensible framework with reusable components that supports the implementation of detectors for a wide variety of indicators.

The output of the DSN is essentially a stream of overheard packets. Hence, it is natural to model the detection framework around the *data stream* abstraction known from database systems [2]. In the remainder of this section we present the particular data stream model we use, the representation of packets as elements of a data stream, and specific operators provided by the framework for analyzing packet traces.

A. Data Streams

A data stream is an unbounded sequence of records. Various data stream management systems (DSMS) have been proposed as generic frameworks to process data streams. We based our framework on PIPES [6], an efficient Java-based infrastructure for processing and exploring data streams that is available under an open-source license.

Three basic abstractions are provided: *sources* that produce data streams, *sinks* that consume data streams, and *operators* that modify data streams. An operator is essentially both a source and a sink. Sinks and operators can *subscribe* to sources and operators, such that a data stream output by the subscribee acts as input for the subscriber. That is, sources, operators, and sinks form a directed *operator graph* with data streams flowing from sources through operators towards sinks. In SNIF, we model each DSN node as a data stream source. An operator graph (being executed on the DSN sink) processes these data streams to detect indicators for problems, and sink nodes inform the user of detected indicators.

B. Records

A data stream record (i.e., an element of the data stream) is a *typed* and *time-stamped* list of attribute-value pairs. The type of a record essentially indicates what

attributes can be found in that record. The time stamp indicates when that record was generated according to a global time scale.

In our implementation, an attribute is a string and each value an object. Two built-in attributes holding record type and time stamp are always available. The DSN produces records of type `Packet` which contain additional attributes holding the contents of an overheard packet (cf. Sect. IV-C). The syntax of the latter attributes follows C syntax for accessing the field of a structure (e.g., `TOS_Msg.addr` in Figure 1). The fields of the encapsulated structures can be accessed recursively (e.g., `TOS_Msg.data.id`). The length of an array can be accessed by appending `.length` to the array field as in `TOS_Msg.data.links.length`.

All attributes referring to packet contents are implemented as *virtual attributes*. Whenever an attribute referring to packet contents is accessed, the packet decoder (cf. Sect. IV-C) is invoked to extract the requested field from the raw packet as captured by the DSN.

C. Basic Operators

Our framework provides a number of basic data stream operators that are sufficient to implement SQL with time windows, but without joins. These operators can be configured by parameters that are either attribute names (prefixed by *attr*), predicates (prefixed by *pred*), or functions over record(s) (prefixed by *func*).

Mapper(attr1Old, attr1New, attr2Old, attr2New, ...): Renames attribute *attrXOld* to *attrXnew* in each record.

ArrayIterator(attrArray): Provides access to array elements by iterating over the array contained in attribute *attrArray*. The operator creates N copies of each input record, where in the i-th copy the array is replaced with array element i. When applied to `LinkAdvertisement.links` in Figure 1, for example, we obtain one record for each pair of neighbors.

Union: Merges the records of all subscribed data streams into one, such that the output data stream has non-decreasing time stamps.

Filter(pred): Drops all records from the data stream for which the given predicate evaluates to false.

TimeWindowAggregator (window, funcAggr, funcHash, attrGroup): Computes an aggregate value over a time window of size *window*. Within the window, records are grouped by the contents of the attribute *attrGroup*. In each group, duplicate records are removed by applying the collision-free hash function *funcHash* to records. If records hash to the same value, only the one with the latest time stamp is kept. Whenever window contents change, the aggregation function *funcAggr* is applied to the window contents. It creates an output record for each group, containing at least an attribute holding the group id and an attribute holding the aggregated value for that group.

Besides common aggregation functions such as counting the number of records as well as computing the sum, average, variance, minimum, and maximum of a given attribute, we provide an aggregation function *ratio(attr) = count / (max(attr) - min(attr) + 1)* with two notable applications. Firstly, when applied to a packet sequence number attribute, *ratio* computes an indicator of observation quality as the fraction of messages sent by a sensor node that can be overheard by the DSN. Secondly, when applied to the time stamp attribute, *ratio* can also be used to estimate congestion (i.e., packets per time unit).

D. Sources

DSNSource: This data stream source is SNIF's interface to the DSN. The individual streams from the DSN nodes are merged into a single stream using the Union operator. Also, duplicate packets (resulting from two or more DSN nodes overhearing the same sensor node) are removed, using a Filter operator with the *predDistinct(window)* predicate that drops a packet if a copy has been observed before within a time window of size *window*.

EmSource: This source uses EmStar link dump files [9] as input, but is otherwise identical to DSNSource.

E. Indicators

This section discusses some operators to support the detection of passive indicators related to nodes, links, and paths (cf. Sect. III). The primary challenge here is to deal with incomplete information due to missing packets the DSN failed to overhear and due to required information required by indicators that is not included in sensor network messages.

Due to space constraints we discuss only operators that are used in later sections of this paper. In addition, SNIF provides operators to measure loss, latency, and congestion that implement the approaches described in Sect. III.

SeqReset(attrSrc, attrSeq, maxSeq): This operator detects a sequence number reset at a node. Parameters are the attribute name holding the source address *attrSrc* and the sequence number *attrSeq*, as well as the maximum sequence number value *maxSeq* before a wrap around. Whenever the earlier of two messages from one sender has a sequence number other than *maxSeq* that is larger than the sequence number of the later message, a record of type `SeqReset` is emitted containing the address of the node.

PacketTracer(attrOrigin, attrDst, predSameFlow): In sensor networks, messages are often not acknowledged at the link layer. In these cases, the sender MAC

address is not included in the message. However, as noted in Sect. III, some indicators require the per-hop source address. The operator PacketTracer reconstructs the per-hop source address for the case that a message is relayed across multiple hops (e.g., a sensor reading being transmitted from a node to the sink). We also assume that all messages of such a flow contain the address of the originator (attribute *attrOrigin*) (e.g., as context information on where a sensor reading has been generated). Also, it must be possible to decide whether two given messages belong to the same flow (e.g., using message contents or an end-to-end sequence number) as implemented by the predicate *predSameFlow*. The per-hop destination address (attribute *attrDst*) must be included in any message to identify the receiver.

To recover the missing source MAC address, the fact is exploited that the per-hop destination address of a message will equal the per-hop source address of the next message in the flow. The operator maintains a table with the last message in each active flow. When a message cannot be associated to a flow in the table using *predSameFlow*, then a new flow is created and the packet is stored. The per-hop source address equals the originator address in this case. Otherwise, if the flow already exists, the per-hop source address of the message equals the per-hop destination address in the packet stored for that flow in the table. A special case are retransmitted packets, where the destination address in the packet equals the destination address stored in the table. The packet in the table is then replaced with the new packet. The operator copies incoming records to the output but appends an attribute holding the discovered source address. Another attribute is added indicating whether or not this packet is a retransmission.

Note that if the DSN fails to overhear one or more of these messages in a row, the destination address of the last overheard packet will be used as the source of the next overheard packet. As a positive side effect of this, we obtain a continuos message flow (i.e., a sequence of messages where the destination address of a message equals the source of the next message in the flow) even if the DSN fails to overhear messages. The operators PathAnalyzer and TopologyAnalyzer operators described below rely on this feature to deal with missing messages.

PathAnalyzer(attrSrc, attrDst, sinkAddr): This operator finds sensor nodes that have a routing path to the sink with MAC address *sinkAddr*. We assume that messages are routed from nodes to the sink along the edges of a spanning tree. Here, a path between a node and the sink exists if a sequence of packets $p_1, ..., p_n$ with increasing time stamps has been observed, such that the source address of p_1 equals the address of the node, the destination address of p_n equals *sinkAddr*, and the destination address of p_i equals the source address of p_{i+1}. The latency of this path is defined as the difference of the time stamps of p_1 and p_n. The names of the attributes holding the source and destination MAC addresses must be given by *attrSrc* and *attrDst*. Note that the above notion of path existence does not imply that packets are actually successfully delivered, but packet loss will result in increased path latency.

To implement this approach, a data structure is maintained that contains for each node A a set $\{(j, t_j)\}$, where j is a node that has a path to A according to the above definition and t_j is the time stamp of the latest message sent by j. We call these nodes j descendants of A. When a message p is observed with source address A, destination address B, and time stamp t, node B is inserted into the data structure if it did not exist before. If B is already listed as a descendant of A then a routing loop exists and B is removed from the A's descendants. Then, the set of descendants of B is updated to include (A, t) and all descendants (j, t_j) of A. Whenever a new descendant is added to the sink or the time stamp of an existing descendant of the sink is updated with a later value, a record with type `GoodPath` is emitted containing the MAC address of the descendant and the latency of the path from the descendant to the sink. In case of a routing loop, a record of type `RoutingLoop` is emitted holding the addresses of sender and receiver of the message causing the loop.

TopologyAnalyzer(window, sinkAddr): This operator implements a heuristic to detect network partitions. Note that partition detection is a non-trivial problem as we do not know the exact set of neighbors of each node. We assume that messages are sent from nodes to the sink (with address *sinkAddr*) along the edges of a spanning tree. Here, a partition is a special case of "no path to sink", where node failures lead to a separation of the node from the sink. In fact, only after a "no path to sink" error has been reported for a node, should the output of TopologyAnalyzer be used to decide whether the reason for this error is caused by a network partition.

To detect such partitions, we construct an approximate view of the network topology. For each node, we maintain a list of recently used (in terms of a time window with length *window*) tree parent nodes by extracting sender and receiver addresses from overheard packets. Also, when a node failure is observed, the respective node is marked as failed in this data structure. A node is considered partitioned if there is no path from that node to the sink in this data structure that uses only nodes that have not failed. In practice, a depth-first search is performed whenever the data structure is modified.

The operator requires two input data streams: a stream of overheard packets and a stream of "node failure" events generated by another operator graph. The output of the operator consists of `Partition` records indi-

cating whether or not a node is currently partitioned from the sink. These records contain a node address, a flag indicating if that node is partitioned or not, and the addresses of any failed upstream nodes that caused the partition.

F. Decision Making

While the above operators are used to detect a variety of passive indicators, it is often not trivial to infer the original problem from these indicators. For example, a primary problem (e.g., node failure) may cause a large number of secondary problems (e.g., routing problems), and we want SNIF to report only the primary problem so as to avoid overwhelming the user with a large set of secondary problems. Below we present operators to help with this.

StateDetector(attrGroup, funcEval): This operator groups records by record type and by the value of the attribute given by *attrGroup*. For each group, the operator stores the latest record. Whenever a new record is inserted into a group, the function *funcEval* is invoked for that group, with the set of stored records as parameter. The evaluation function outputs a record of type `State`, containing the value of the grouping attribute as well as the current state of the group. A typical application of this operator is to compute the current state of each node (e.g., ok, crashed, no route, ...). Here, records are grouped by node address. See Sect. VI for an example.

predStateChange(attrGroup, attr1, attr2, ...): This predicate is used with the Filter operator to remove duplicate records. Records are first grouped by *attrGroup*. In each group, a record is dropped unless it differs from the previous record in that group in at least one of the attributes *attrX*. When applied to the output of StateDetector, (node) state changes can be computed.

G. Discussion

Currently, using SNIF requires to write small amounts of Java code to configure operators and to connect them to form an operator graph. PIPES provides a simple GUI to perform the latter interactively without writing code. We plan on extending this tool to allow interactive construction of "debuggers" based on SNIF.

We also considered the use of SQL dialects for data streams (e.g., [7], [24]).[2] However, SNIF mainly consists of *stateful* operators which cannot be easily expressed with these dialects. Also, typical operator graphs contain long chains of these stateful operators, which would require bulky, deeply nested SQL subqueries.

[2]In fact, the operators described in Sect. V-C are a minimalist implementation of a streaming SQL.

VI. A SENSOR NETWORK DEBUGGER

In this section we show how SNIF can be used to construct a debugger that can detect a variety of common problems with a sensor network. The debugger is intended for a typical data gathering application, where nodes send sensor readings at regular intervals along a spanning tree to a sink. Note, however, that this is only one example use of SNIF (although a typical one), rather than being *the* debugger for *the* sensor network application. We first describe the chosen application in more detail, before turning to the details of the debugger.

A. Sensor Network Model

Despite visions of sensor networks with complex in-network data aggregation, most sensor networks deployed to date are simple data gathering applications[3] (e.g., [26], [31], [15]), where nodes send raw sensor readings at regular intervals along a spanning tree across multiple hops to a sink.

Two prominent implementations of this scheme are the he beacon-based Multihop routing service of the Extensible Sensing System (ESS) [12] and applications based on MintRoute [33]. They both implement a similar multi-hop routing scheme as described below. Our example debugger is tailored to ESS, but could be easily adopted to work with MintRoute or other similar applications.

All nodes broadcast *beacon messages* at regular intervals. To discover neighbors, nodes overhear these messages and estimate the quality of incoming links to the neighbors based on message loss. Nodes then broadcast *link advertisement messages* at regular intervals, containing a list of neighbors and link quality estimates. Overhearing these messages, nodes compute the bidirectional link quality to decide on a good set of neighbors. To construct a spanning tree of the network with the sink at the root, nodes broadcast *path advertisement messages*, containing the quality of their current path to the sink. Nodes overhearing these messages can then select the neighbor with the best path as their parent and broadcast an according path advertisement message. All this is executed continuously to adapt neighbors and paths to changing network conditions. Finally, *data messages* are sent from nodes to the sink along the edges of the spanning tree across multiple hops.

In the ESS Multihop protocol, all messages except data messages are broadcast messages and contain per-hop source address. Data messages contain the address of the originator of the sensor data and the per-hop destination address, but not the per-hop source address. In addition, beacon messages and data messages contain a sequence number.

[3]One reason for this is that many deployments are scientific applications, and scientists typically want to see as much data as possible.

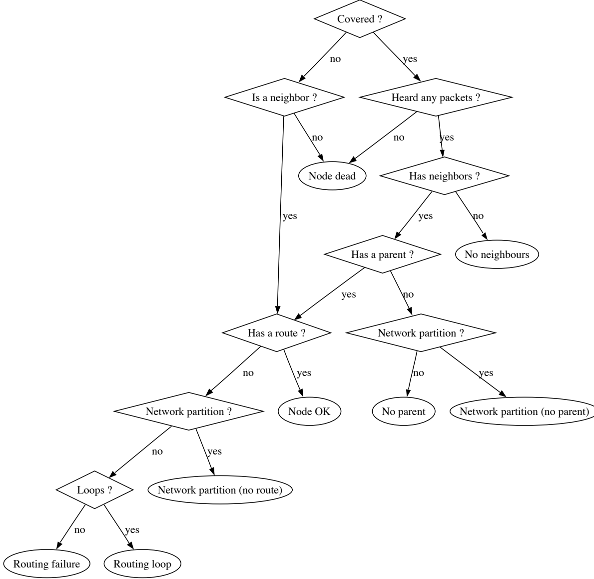

Fig. 2. Node state decision tree.

B. Debugger

Our example debugger can detect a set of pre-defined *problem events*. Each such event identifies a problem and refers to a node in the network. An event notification is represented as a data stream record containing the address of the affected node, the type of event, and possibly event-specific additional information. In particular, the following events are detected: *node ok*, *node death*, *node reboot*, *no neighbors*, *no parent*, *partitioned from sink*, *path to sink loops*, and *no path to sink*.

Node reboot events are detected by filtering DSNSource for beacon packets and applying the SeqReset operator. The detection of the remaining events is based on the binary decision tree depicted in Figure 2. The leaves of this tree represent possible states of a node. Whenever the state of a node changes, a respective event is generated for this node.

The binary decision tree is implemented using the StateDetector operator. The output records describing node states are filtered for state changes using the predStateChange filter predicate. Each decision in the tree requires an operator graph that extracts the required information from the stream of observed packets. Below we describe how each of these decisions is implemented with an operator graph. Note that the individual operator graphs described below partially overlap. These common subgraphs are instantiated only once.

Covered?: This test examines whether a sensor node can be observed with sufficient quality by the DSN by examining the percentage of beacon messages that have been received from this node. To implement this test, DSNSource is filtered for beacon messages. The stream of beacon messages is then fed to a TimeWindowAggregator. Using the ratio aggregation function applied to the sequence number of the beacon messages, the fraction of received beacons per node is computed. The test succeeds for a node if the fraction for this node is above a given threshold.

Heard any packets?: This test succeeds if any packet from a sensor node could be overheard. Since data messages do not contain the per-hop source address, DSNSource is filtered for data packets and PacketTracer is applied to reconstruct the source address. Also, DSNSource is filtered for the remaining packet types (beacon, link and path advertisements) that do already contain the per-hop source address. The resulting data streams are merged with the Union operator to obtain a stream of all packets containing source addresses. This stream is then fed to a TimeWindowAggregator to count the number of packets per node using the count aggregation function. The test succeeds for a node if at least one packet was heard from this node.

Is a neighbor?: This test checks whether a sensor node is listed as a neighbor of any other node in the network. DSNSource is filtered for link advertisement packets. Since each link advertisement contains an array of neighbors, the ArrayIterator operator is used to create one record for each node being listed as a neighbor. Using TimeWindowAggregator with the count aggregation function we obtain the number of times a node is listed as a neighbor. The test succeeds for a node, it it was listed as a neighbor at least once.

Has any neighbors?: This test examines whether a node has any neighbors. DSNSource is filtered for link advertisement packets containing at least one neighbor. Using TimeWindowAggregator, the number of such advertisements per node is computed. The test succeeds for a node if at least one non-empty link advertisement was heard from this node.

Has a parent?: This test examines whether a node has a parent in the tree. DSNSource is filtered for path advertisement packets. Using TimeWindowAggregator, the number of such advertisements per node is computed. The test succeeds for a node if at least one path advertisement was heard from this node.

Has a route?: This test checks whether a node recently had a routing path to the sink. DSNSource is filtered for data messages. PacketTracer is applied to reconstruct the source address. PathAnalyzer is applied and its output filtered for good route reports. Using TimeWindowAggregator, the number of good route reports per node is counted. The test succeeds for a node if a good route was reported at least once for this node.

Loops?: This test checks whether the path from a node to the sink recently had any loops. DSNSource is filtered for data messages. PacketTracer is applied to reconstruct the source address. PathAnalyzer is applied

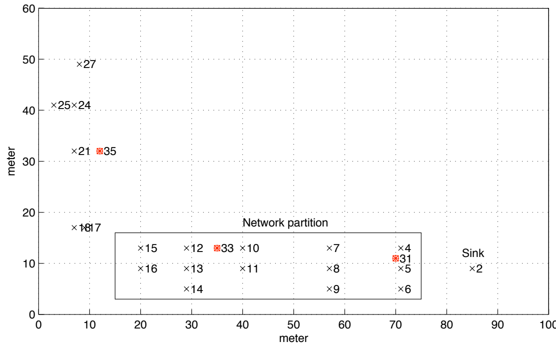

Fig. 3. Experiment setup: WSN (2-27) and DSN (31-35).

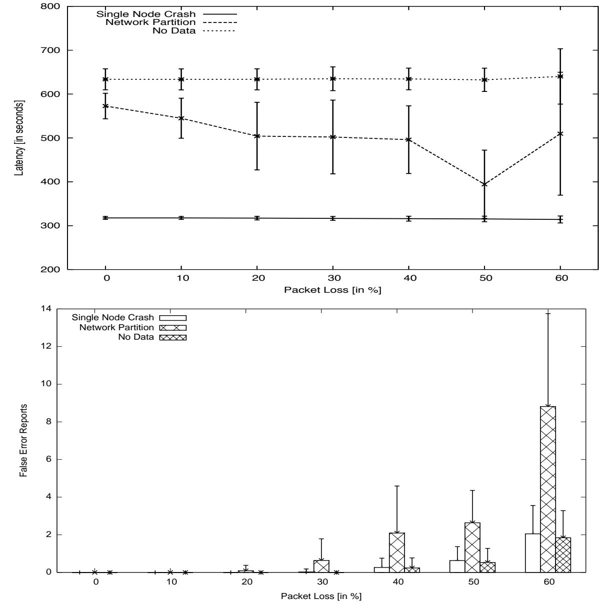

Fig. 4. Reporting latency and number of false reports as a function of packet loss.

and its output filtered for routing loop reports. Using TimeWindowAggregator, the number of good route reports per node is counted. The test succeeds for a node if a routing loop was reported at least once for this node.

Network partition?: This test checks if a bad path from a node to the sink was caused by a network partition. DSNSource is filtered for data messages. PacketTracer is applied to reconstruct the source address. TopologyAnalyzer is applied to detect partitions. TopologyAnalyzer is also subscribed to the output of StateDetector in order to obtain *node death* events. The test succeeds for a node if the last record received from TopologyAnalyzer says that this node is partitioned.

VII. Evaluation

To evaluate SNIF, we performed a series of experiments with the debugger presented in Sect. VI. Even though the debugger is only one example use of SNIF, we consider it a representative application as it can detect many of the problems described in Sect. II. We used the experimental setup described in [20]. In this setup, the Extensible Sensing System (ESS) [12], which is also used in real-world deployments, is executed in the EmStar emulator [9]. The Multihop routing protocol of ESS is described in Sect. VI-A, with beacons and link advertisements being sent every 10 seconds and path advertisements every 80 seconds (resulting in a tree update every 80 seconds). The network is time-synchronized and each node sends sensor readings to the sink every 30 seconds. A neighbor is considered gone if no beacons have been received for 320 seconds.

As depicted in Fig. 3, we consider a network of 21 nodes forming a multi-hop topology with a diameter of 7 hops. Node 2 acts as the sink. We added three DSN nodes (nodes 31, 33, and 35 marked with squares in Fig. 3). The link dump files of the DSN nodes were used as input to the debugger. Since some sensor nodes could be overheard by more than one DSN node, the DSN received 1.3 ± 0.5 copies of each sensor network message during the experiments, while 4% of the beacon messages were lost (i.e., not overheard by any DSN node).

A. Accuracy and Latency

In order to evaluate the accuracy (number and type of false error reports) and latency (time between failure injection and report) of our debugging tool, we run a set of experiments injecting three types of faults into the network: node failure, network partition, and no data. Furthermore, to evaluate the performance and robustness of our tool under more realistic conditions, we introduced additional loss into the DSN by randomly discarding overheard messages. The duration of each experiment was 30 minutes with faults being injected randomly between 10 and 15 minutes after experiment begin. We report averages and standard deviation over multiple runs. Fig. 4 shows the results on the latency (top) and the number of false error reports (bottom) for each of the faults injected into the network with respect to the percentage of introduced packet loss from 0% to 60%.

In the first experiment, we performed 40 runs and injected a single node failure per run, such that all nodes but the sink failed twice. All node crashes were correctly detected and no false errors were reported. The latency of the reports is mainly determined by the size of the time window used to implement the *Heard any packets?* test. We used 320 seconds (the same value used by ESS to detect gone neighbors as noted above). As beacon messages are sent every 10 seconds, we expected the latency to be between 310 and 320 seconds. We then introduced additional loss by uniformly dropping a given

percentage of packets from the DSN output. Observe that latency is not significantly affected by the loss of packets. The number of false positives is neglectable until %30 of packet loss and and raises significantly with more than 50% as depicted in Fig. 4 (bottom). We analyzed the generated error reports and observed that for up to 70% of packet loss, we only observed *no neighbor* and *no parent* reports. These reports are caused by missing link and path advertisements respectively which are rarely sent. For higher packet loss, we found *node dead* reports for working nodes. We never observed any false negatives.

In the second experiment we made nodes 4-16 fail at random times to partition nodes 17-27 from the remainder of the network. We would expect a *network partition* error for nodes 17-27 with some of the nodes 4-16 reported as the cause for the partition. We report the latency until the first node was classified as partitioned. The latency for the network partition detection is determined by the window sizes of the *Has a parent?* and *Has a route?* tests which are set to 640 seconds (twice as much as for *Heard any packets?*). As *Has a route?* basically tracks multi-hop packets which are sent often, it reacts with shortly before 640 seconds. The *Has a parent?* tests fails, if no path announcements were observed during the time window. As path announcements are only sent every 80 seconds, missed packets cause a drop in the average latency down to around 572 seconds. This also explains the latency decrease with higher packet loss.

In the third experiment, we injected faults into the Multihop routing component of single nodes such that an affected node stops sending data messages (path problem), while still broadcasting beacons and advertisements (no link or node problems). We would expect a *no route* error for the affected node and all other nodes whose paths contain the former. We report the time until the affected node is marked with *no route*. In this experiment, the latency is determined by the window size of the *Has a route?* test which is 640 seconds. Again, missed packets cause a drop in the latency but as most nodes in the network forward packets for other nodes and data packets are at least sent every 30 seconds, the average latency should be close to 640 seconds. The average of 633 ± 24 seconds confirms this.

B. SNIF Performance

We also studied the performance overhead of SNIF itself. During one 30 minute experiment run without any fault injections, the DSN collected 261 kB of data, resulting in an average data rate of 1.2 kbps including duplicate packets. Note that this equals about 0.3% of the effective Bluetooth 1.2 bandwidth of 400 kbps. SNIF was executing on a 2 GHz PC using Java 1.5. The total cpu time for processing the above amount of data was about 13 seconds, which equals about 0.7% of the experiment duration of 30 minutes.

C. A Real Bug

In the course of our experiments, we encountered a bug in ESS Multihop. At one point we decided to upgrade to a new version of EmStar that fixed a bug with collision handling. After the upgrade, we suddenly observed a large number of *no parent* error reports without injecting any faults. By examining the source code of Multihop, we learned that nodes react to receipt of a path advertisement message by updating their parent selection and broadcasting their updated path advertisement immediately without any delay. Here, the original path advertisement broadcast results in an implicit synchronization of all receivers, such that the secondary path advertisements collide with high probability without being retransmitted. By adding a random jitter, we were able to fix this problem.

VIII. Related Work

Most closely related to SNIF is work on active debugging of sensor networks, notably Sympathy [20] and Memento [22]. However, both systems require instrumentation of sensor nodes and introduce monitoring protocols in-band with the actual sensor network traffic. Also, both tools only support a fixed set of problems, while SNIF provides an extensible framework.

Tools for sensor network management such as NUCLEUS [30] provide read/write access to various parameters of a sensor node that may be helpful to detect problems. However, this approach also requires active instrumentation of the sensor network.

Complementary to SNIF is work on simulators (e.g., SENS [25]), emulators (e.g., TOSSIM [16]), and testbeds (e.g., MoteLab [32]) as they support development and test of sensor networks *before* deployment in the field. In particular, testbeds typically provide a wired backchannel from each node, such that sensor nodes can be instrumented to send status information to an observer. EmStar [9] integrates simulation, emulation, and testbed concepts into a common framework where some nodes physically exist in a testbed or in the real world, while the majority of nodes is being emulated or simulated. Physical nodes need instrumentation and a wired backchannel.

Passive observation by means of packet sniffing has also been applied to wireless (and wired) LANs [13]. However, sensor networks differ substantially from wireless LANs. While typical wireless LANs are single-hop networks that can be observed with one or few sniffers, sensor networks are typically multi-hop networks. Also, many of the problems encountered during deployment of sensor networks are not present in WLANs. Also,

most applications of sniffing to (W)LANs log overheard traffic for offline analysis, while SNIF performs online analysis.

In the more general context of management and debugging of distributed systems, a large body of related work exists. Due to space constraints, we limit our discussion to very closely rated work. One such class of closely related work is performance debugging of distributed systems (e.g., [1], [3]) where message traces are used to reconstruct causality paths and their latencies. While in principle applicable to sensor networks, these approaches are narrowly focused on a very specific problem and analysis is performed offline. In contrast, we provide a framework for online traffic analysis. A number of data stream management systems have been specifically developed for network traffic analysis (e.g., [7], [24]). However, as mentioned earlier, we found it difficult if not impossible to express stateful SNIF operators using the SQL variants of these systems.

IX. Conclusions

We showed that many problems commonly encountered during the deployment of a sensor network can be detected without instrumentation of the sensor network by overhearing and analyzing sensor network traffic. We implemented this approach by providing a distributed network sniffer and a framework for online analysis of the resulting streams of overheard network messages. Both, the sniffer and the framework can be used with any WSN that uses a compatible radio. We also demonstrated an application of this framework to debug an existing data gathering application. It could be shown that our tool supports accurate and timely detection of problems with this application even if the sniffer fails to overhear a large fraction of the sensor network traffic. Using this debugger we found an actual bug in the application.

Future work includes the development of user interface that allows easier construction of debuggers for specific applications. Currently, small amounts of Java code must be written to form an operator graph. Another worthwhile direction would be the execution of data streams operators (e.g., to detect node and link problems) on DSN nodes to reduce the bandwidth usage of the DSN. Currently, DSN nodes deliver all overheard messages to the DSN sink, where analysis is also performed. Finally, we plan to support semi-passive observation in future versions of our framework. SNIF code will be released at [36].

References

[1] M. K. Aguilera et al. Performance debugging for distributed systems of black boxes. In *SOSP 2003*.

[2] B. Babcock et al. Models and issues in data stream systems. In *PODS 2002*.

[3] P. T. Barham et al. Using magpie for request extraction and workload modelling. In *ODSI 2004*.

[4] J. Beutel et al. Scalable topology control for deployment-sensor networks. In *IPSN '05*.

[5] P. Buonadonna et al. Task: Sensor network in a box. In *EWSN 2005*.

[6] M. Cammert et al. Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources. Technical report, University of Marburg, 2003.

[7] C. Cranor et al. Gigascope: A Stream Database for Network Applications. In *SIGMOD 2003*.

[8] D. Ganesan et al. Complex Behavior at Scale: An Experimental Study of Low-Power Wireless Sensor Networks. Technical Report CSD-TR 02-0013, UCLA, 2002.

[9] L. Girod et al. EmStar: A software environment for developing and deploying wireless sensor networks. In *USENIX 2004*.

[10] J. Gray. Why do Computers Stop and What Can be Done About it? In *5th Symp. on Reliability in Distributed Software and Database Systems*, 1986.

[11] B. Greenstein et al. A sensor network application construction kit (snack). In *Sensys 2004*.

[12] R. Guy et al. Experiences with the extensible sensing system ess. Technical Report 61, CENS, 2006.

[13] T. Henderson et al. Measuring wireless LANs. In R. Shorey, A. L. Ananda, M. C. Chan, and W. T. Ooi, editors, *Mobile, Wireless, and Sensor Networks*. Wiley, 2006.

[14] Chalermek Intanagonwiwat et al. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1), 2003.

[15] K. Langendoen et al. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *WPDRTS 2006*.

[16] P. Levis et al. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Sensys 2003*.

[17] A. Mainwaring et al. Wireless sensor networks for habitat monitoring. In *WSNA '02*.

[18] P. Padhy et al. Glacial environment monitoring using sensor networks. In *REALWSN' 05*.

[19] J. Polastre et al. Analysis of wireless sensor networks for habitat monitoring. In *Wireless Sensor Networks*, chapter 18. 2004.

[20] N. Ramanathan et al. Sympathy for the sensor network debugger. In *SenSys '05*.

[21] N. Ramanathan et al. Towards a Debugging Systems for Sensor Networks. *Int. J. Network Management*, 15:223–234, 2005.

[22] S. Rost et al. Memento: A Health Monitoring System for Wireless Sensor Networks. In *SECON 2006*.

[23] A. Sobeih et al. Finding bugs in network protocols using simulation code and protocol-specific heuristics. volume 3785 of *LNCS*, 2005.

[24] M. Sullivan et al. Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX 1998*.

[25] S. Sundresh et al. SENS: A Sensor, Environment and Network Simulator. In *Annual Simulation Symposium 2004*.

[26] R. Szewcyk et al. An analysis of a large scale habitat monitoring application. In *Sensys 2004*.

[27] R. Szewczyk et al. Lessons from a Sensor Network Expedition. In *EWSN 2004*.

[28] J. Tateson et al. Real world issues in deploying a wireless sensor network for oceanography network for oceanography. In *REALWSN'05*.

[29] S. Tilak et al. A Taxonomy of Wireless Micro-Sensor Network Models. *MC2R*, 6(2):28–36, April 2002.

[30] G. Tolle et al. Design of an application-cooperative management system for wireless sensor networks. In *EWSN 2005*.

[31] G. Tolle et al. A macroscope in the redwoods. In *SenSys '05*.

[32] G. Werner-Allen et al. Motelab: a wireless sensor network testbed. In *IPSN 2005*.

[33] A. Woo et al. Taming the underlying challenges if reliable multihop routing in sensor networks. In *Sensys 2003*.

[34] Berkeley Motes. www.xbow.com/Products/Wireless_Sensor_-Networks.htm.

[35] BTnodes. www.btnode.ethz.ch.

[36] SNIF. www.vs.inf.ethz.ch/res/show.html?what=snif.