SNIF: A Comprehensive Tool for Passive Inspection of Sensor Networks

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

ABSTRACT

Deployment of sensor networks in real-world settings is a labor-intensive and cumbersome task: environmental influences often trigger problems that are difficult to track down due to limited visibility of the network state. In this extended abstract, we summarize our ongoing efforts to develop a tool for passive inspection of sensor networks, where the network state can be inferred without instrumentation of sensor nodes. We also discuss next steps to make this tool applicable to a larger class of applications.

1. INTRODUCTION

Deployment of sensor networks in real-world settings is typically a labor-intensive and cumbersome task (e.g. [5, 6, 12, 13, 15]). While simulation and lab testbeds are helpful tools to test an application prior to deployment, they fail to provide realistic environmental models (e.g., regarding radio signal propagation, sensor stimuli, chemical/mechanical strain on sensor nodes). Hence, environmental effects often trigger bugs or degrade performance in a way that could not be observed during pre-deployment testing. To track down such problems, a developer needs to inspect the state of network and nodes. While this is easily possible during simulation and experiments on lab testbeds (wired backchannel from every node), access to network and node states is very constrained after deployment.

Current practice to inspect a deployed sensor network requires *active* instrumentation of sensor nodes with monitoring software. Monitoring traffic is sent in-band with the sensor network traffic to the sink (e.g., [6, 11, 14]). Unfortunately, this approach has several limitations. Firstly, problems in the sensor network (e.g., partitions, message loss) also affect the monitoring mechanism, thus reducing the desired benefit. Secondly, scarce sensor network resources (energy, cpu cycles, memory, network bandwidth) are used for inspection. Thirdly, the monitoring infrastructure is tightly interwoven with the application. Hence, adding/removing instrumentation may change the application behavior in subtle ways, causing probe effects. Also, it is non-trivial to adopt the instrumentation mechanism to different applications.

In contrast to the above, we propose a *passive* approach for sensor network inspection by overhearing and analyzing sensor network traffic to infer the existence and location of typical problems encountered during deployment. To overhear network traffic, a so-called *deployment support network* (DSN) [1] is used: a wireless network that is temporarily installed alongside the actual sensor network during the deployment process (see Fig. 1). 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 network among the DSN nodes to reliably collect overheard packets. A data stream

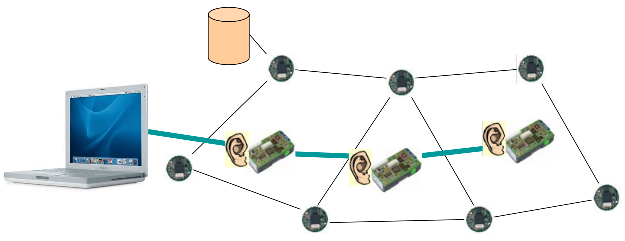


Figure 1: A deployment-support network (rectangular nodes) is a physical overlay network that overhears sensor network (round nodes) traffic and delivers it to a sink using a second radio.

framework performs online analysis of the resulting packet stream to infer and report problems soon after their occurrence.

This approach removes the above limitations of active inspection: no instrumentation of sensor nodes is required, sensor network resources are not used. The inspection mechanism is completely separated from the application, can thus be more easily adopted to different applications, and can be added and removed without altering sensor network behavior.

So far, we analyzed and classified problems typically found during deployment [7], implemented a basic version of the Sensor Network Inspection Framework (SNIF) [10], and conducted a case study on data gathering applications [9] to demonstrate the feasibility and benefits of our approach. Although possible in principle, it is currently difficult to adopt SNIF to application types other than data gathering application. Further work is needed to add this missing flexibility. In the remainder of this abstract, we give an overview of SNIF's architecture and discuss next steps.

2. PASSIVE INSPECTION OF DATA GATHERING APPLICATIONS

SNIF supports the detection of specific *problems* (e.g., node reboot) that occur during deployment of a sensor network. For this, we first need to identify the problems that can occur and that should be detected by SNIF. Secondly, we need to provide *passive indicators* for each of the problems, which allow to infer the existence of a problem from observed packet traces. For this, one needs to analyze the message protocols used in the sensor network. Finally, SNIF needs to be configured to implement these passive indicators.

In our work, we focus on so-called *data gathering applications* (e.g., [12, 15]), where nodes send raw sensor readings at regular intervals along a spanning tree across multiple hops to a sink. The reason for our choice is that almost all existing non-trivial deployments are data gathering applications. Below, we will first characterize data gathering applications in more detail, before presenting typical problems with these applications and matching passive indicators.

2.1 Application Model

Systems for data gathering such as the Extensible Sensing System (ESS) [4] need to maintain a spanning tree of the network along which sensor values are routed to the sink. To support neighbor discovery, all nodes broadcast *beacon messages* at regular intervals. Each beacon message contains a sequence number. To discover neighbors, nodes overhear these messages and estimate the quality of incoming links from 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. Finally, *data messages* are sent from nodes to the sink along the edges of the spanning tree across multiple hops.

2.2 Problems and Indicators

A indicator is an observable behavior of a sensor network that hints (in the sense of a heuristic) the existence of a specific problem. We are interested in *passive* 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.

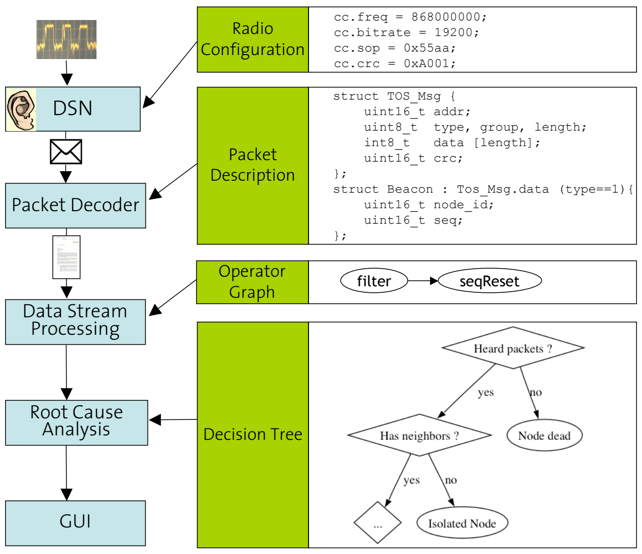
In [7] we studied existing deployments to identify common problems and derived passive indicators for them. We classify problems according to the number of nodes involved into four classes based on existing deployments: *node problems* that involve only a single node, *link problems* that involve two 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. Below, we provide for each category an exemplary problem and passive indicators to detect it.

Node reboot, as an example for a node problem, causes the sequence number counter of the affected node to be reset to an initial value (typically zero). Hence, the sequence number contained in beacon messages sent by the node will jump to a smaller value after a reboot with high probability even in case of lost messages, which can serve as an indicator for reboot.

An *isolated node*, as an example for a link problem, has no neighbors in the network topology. An indicator for this problem is that the node is not listed in any link advertisement messages send by other nodes.

An *orphaned node*, as an example for a path problem, has no parent in the routing tree. Such nodes will either send no path announcement messages at all or path announcements contain an infinite distance to the sink (depending on the protocol details), which can be used as a passive indicator.

A *partitioned node*, as an example of a global problem, is disconneted from the sink, for example due to death of a node on the path. A node is considered as partitioned if all paths from the node to the sink involve dead nodes. This predicate requires an approximate view on the network topology which is reconstructed on the



```
cc.freq = 868000000;
cc.bitrate = 19200;
cc.sop = 0x55aa;
cc.crc = 0xA001;
```

```
struct TOS_Msg {
    uint16_t addr;
    uint8_t  type, group, length;
    int8_t   data [length];
    uint16_t crc;
};
struct Beacon : Tos_Msg.data (type==1){
    uint16_t node_id;
    uint16_t seq;
};
```

Figure 2: Architecture of SNIF.

base of observed data packets. Periodic checks on the reconstructed topology serve as a passive indicator here.

3. SNIF

In this section we outline how passive indicators discussed in the previous section can be implemented in SNIF. For this, consider the architecture of SNIF as depicted in Fig. 2, which consists of a deployment support network to overhear sensor network traffic, a packet decoder to access the contents of overheard packets, a data stream processor to analyze packet streams for problems, a decision tree to infer the state of each sensor node, and a user interface to display these states. The key design goal for SNIF is generality that is, it should support passive inspection of a wide variety of sensor network protocols and applications. Below we give an overview of these components. More details can be found in a [10].

3.1 Deployment Support Network (DSN)

To overhear the traffic of multi-hop networks, multiple radios are needed, forming a distributed network sniffer. We use a so-called deployment support network for this purpose, a wireless network of DSN nodes, each of which provides two radios.

Our current implementation of a DSN is based on the BTnode Rev. 3 [2], 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 (e.g., also used on MICA2) which is used as the WSN radio. Using a scatternet formation algorithm, the DSN nodes form a robust Bluetooth scatternet (see [1] for details). A laptop computer with Bluetooth acts as the SNIF sink that connects to a nearby DSN node. This DSN node thereupon acts as the DSN sink and forms the root of an overlay tree spanning the whole DSN. The SNIF sink can send data to DSN nodes down the tree while DSN nodes send overheard packets up the tree to the sink.

Time synchronization exploits the fact that Bluetooth uses a TDMA MAC protocol and thus performs clock synchronization internally, providing an interface to read the Bluetooth clock and its offset to the clocks of network neighbors. We use this interface to compute the clock offset of each DSN node to the DSN sink. A detailed description of our time synchronization protocol can be found in [8].

3.2 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. Our generic PHY implementation supports configurable carrier frequency, baud rate, and checksumming details as illustrated in Fig. 2. Regarding MAC, we exploit the fact that – regardless of the specific MAC protocol used – a radio packet always has to be preceded by a preamble and a start-of-packet (SOP) delimiter to synchronize sender and receiver. In our generic MAC implementation, every DSN node has its WSN radio turned to receive mode all the time, looking for a preamble followed by the SOP delimiter in the received stream of bits. Once an SOP has been found, payload data and a CRC follow. This way, DSN nodes can receive packets independent of the actual MAC layer used.

3.3 Packet Decoder

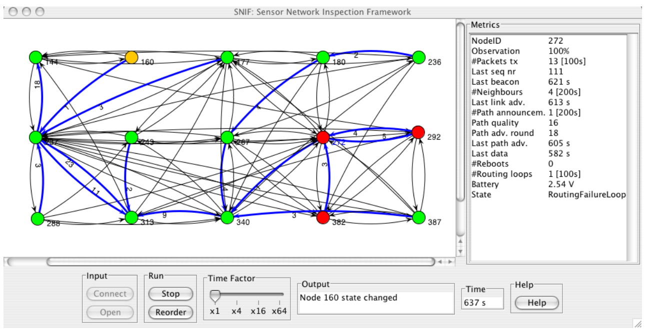
Again, since no standard protocols exist for sensor networks, we need a flexible mechanism to decode overheard packets. Since most programming environments for sensor nodes are based on the C programming language or a dialect of it (e.g., nesC for TinyOS), it is common to specify message contents as (nested) C structs in the source code of the sensor network application. Our 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 of the packet decoder 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.

Fig. 2 shows an example of a TinyOS message (`TOS_Msg`) holding a beacon data unit if the message type of the TinyOS message equals 1. The result of packet decoding is a record consisting of a list of name-value pairs, where each pair holds the name and value of a data field in the packet.

3.4 Data Stream Processor

The resulting stream of packets is then fed to a data stream processor to detect any problems with the sensor network. The data stream processor executes operators that take a stream of records as input and produce a different stream of records as output. The output of an operator can be connected to the input of other operators, resulting in a directed operator graph. SNIF provides a set of standard operators, e.g., for filtering, aggregation over time windows, or merging of multiple streams into one. In addition, application-specific operators to detect specific problems in the sensor network may be required. Fig. 2 shows an simple operator graph that is used to detect node reboots as described in Sect. 2.2. The first operator (`filter`) reads the packet stream generated by the DSN and removes all packets that are not beacon packets. The second operator (`seqReset`) remembers the last sequence number received from each node and checks if a newly received sequence number is smaller than the previous one for this node, in which case the node has rebooted unless there was a sequence number wrap-around (i.e., maximum sequence number has been reached and sequence counter wraps to zero).



NodeID	272
Observation	100%
#Packets tx	13 [100s]
Last seq nr	111
Last beacon	621 s
#Neighbours	4 [200s]
Last link adv.	613 s
#Path announcem.	1 [200s]
Path quality	16
Path adv. round	18
Last path adv.	605 s
Last data	582 s
#Reboots	0
#Routing loops	1 [100s]
Battery	2.54 V
State	RoutingFailureLoop

```
Node 160 state changed
```

Time: 637 s

Figure 3: An instance of SNIF's user interface.

3.5 Root Cause Analysis

The next step is to derive the state of each sensor node, which can be either "node ok" or "node has problem X". Note that the operator graphs mentioned above may concurrently report multiple problems for a single node. In many cases, one of the problems is a consequence of another problem. For example, a node that is dead also has a routing problem. In such cases, we want to report only the primary problem and not secondary problems. For this, we use a decision tree, where each internal node is a decision that refers to the output of an operator graph, and each leaf is a node state. In the example tree depicted in Fig. 2, we first check (using the output of an operator graph that counts packets received during a time window) if any messages have been received from a node. If not, then the state of this node is set to "node dead". Otherwise, if we received packets from this node, we next check if this node has any neighbors (using an operator graph that counts the number of neighbors contained in link advertisement packets received from this node). If there are no neighbors, then the node state is set to "node isolated". Here, the check for node death is above the check for isolation in the decision tree, because a dead node (primary problem) is also isolated (secondary problem).

3.6 User Interface

Finally, node states and additional information are displayed in the graphical user interface. The core abstraction implemented by the user interface is a network graph, where nodes and links can be annotated with arbitrary information. The user interface also supports recording and playback of executions. A snapshot of an instance of the user interface is shown in Fig. 3. Here, node color indicates state (green: ok, gray: not covered by DSN, yellow: warning, red: severe problem), detailed node state can displayed by selecting nodes. Thin arcs indicate what a node believes are its neighbors, thick arcs indicate the paths of multi-hop data messages.

4. RELATED WORK

Most closely related to SNIF is work on active debugging of sensor networks, notably Sympathy [6] and Memento [11]. 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 [14] 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.

5. NEXT STEPS

As mentioned in Sect. 2.1, our current work is focused on data gathering applications. As other types of applications such as tracking and event detection are deployed, we will analyze experiences gained from deployments and add support for inspection of these applications to SNIF. For this, novel indicators may have to be implemented in SNIF. While SNIF supports this flexibility in principle through composition and parametrization of data stream operators, currently Java code needs to be written and the developer has to be familiar with SNIF internals. One of the next steps is therefore the development of appropriate high-level specification techniques to support more convenient configuration of SNIF for different types of applications. In particular, we envision a graphical notation, allowing a user to devise these specifications using a graphical user interface.

Ultimately, we want to achieve (semi-)automatic generation of these specifications from application programs. For this, we will work on analyzing high-level declarative program specifications such as SNlog [3]. These capture the application semantics in a more direct way than procedural programs, such that it may be possible to derive SNIF configurations without explicit annotations.

6. ACKNOWLEDGMENTS

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.

7. REFERENCES

[1] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable Topology Control for Deployment-Sensor Networks. In *IPSN 2005*.

[2] BTnodes. A Distributed Environment for Prototyping Ad Hoc Networks. www.btnode.ethz.ch.

[3] D. C. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. Technical Report UCB/EECS-2006-132, EECS Department, UC Berkeley, October 2006.

[4] R. Guy, B. Greenstein, J. Hicks, R. Kapur, N. Ramanathan, T. Schoellhammer, T. Stathopoulos, K. Weeks, K. Chang, L. Girod, and D. Estrin. Experiences with the Extensible Sensing System ESS. Technical Report 61, CENS, 2006.

[5] P. Padhy, K. Martinez, A. Riddoch, H. L. R. Ong, and J. K. Hart. Glacial Environment Monitoring using Sensor Networks. In *REALWSN 2005*.

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

[7] M. Ringwald and K. Römer. Deployment of Sensor Networks: Problems and Passive Inspection. In *WISES 2007*.

[8] M. Ringwald and K. Römer. Practical Time Synchronization for Bluetooth Scatternets. In *BROADNETS 2007*.

[9] M. Ringwald, K. Römer, and A. Vialetti. Passive Inspection of Sensor Networks. In *DCOSS 2007*.

[10] M. Ringwald, K. Römer, and A. Vialetti. SNIF: Sensor Network Inspection Framework. Technical Report 535, Departement of Computer Science, ETH Zurich, 2006.

[11] S. Rost and H. Balakrishnan. Memento: A Health Monitoring System for Wireless Sensor Networks. In *SECON 2006*.

[12] R. Szewcyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An Analysis of a Large Scale Habitat Monitoring Application. In *SenSys 2004*.

[13] J. Tateson, C. Roadknight, A. Gonzalez, S. Fitz, N. Boyd, C. Vincent, and I. Marshall. Real World Issues in Deploying a Wireless Sensor Network for Oceanography. In *REALWSN 2005*.

[14] G. Tolle and D. Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *EWSN 2005*.

[15] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A Macroscope in the Redwoods. In *SenSys 2005*.