Passive Inspection of Sensor Networks

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— **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 paper we present a framework for passive inspection (i.e., no instrumentation of sensor nodes required) of deployed sensor networks and show how this framework can be used to inspect data gathering applications. The basic approach is to temporarily install a distributed network sniffer alongside the inspected sensor network, with overheard messages being analyzed by a data stream processor and network state being displayed in a graphical user interface. Our tool can be flexibly applied to different sensor network operating systems and protocol stacks, and can deal well with incomplete information.**[1]

I. Introduction

Deployment of sensor networks in real-world settings is typically a labor-intensive and cumbersome task [4], [10], [16], [17], [18], [19], [25], [26], [28]. 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 and monitoring traffic is sent in-band with the sensor network traffic to the sink (e.g., [19], [22], [27]). 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. In Sympathy [19], for example, up to 30% of the network bandwith is used for monitoring traffic. 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 nontrivial to adopt the instrumentation mechanism to different applications. For example, [19], [22] assume a certain tree routing protocol being used by the application and reuse that protocol for delivering monitoring traffic.

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) [3] is used: 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 network among the DSN nodes to reliably collect overheard packets. A data stream framework performs online analysis of the resulting packet stream to infer and report any problems soon after their occurrence.

[1]The work presented in this paper was partially supported by the by the Swiss National Science Foundation under grant number 5005-67322 (NCCR-MICS).

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. Online analysis (as opposed to long periods of data collection followed by offline analysis) contributes to a more effective deployment process, as it allows an engineer to go out and study affected nodes while a problem is still present. Also, problems can be fixed in an incremental fashion as they occur, thus reucing the chance for complex aftereffects. Besides these advantages, we need to address a number of challenges:

Incomplete information: The DSN may fail to overhear some packets and messages might not contain all information that is needed to infer a problem. To support robust problem detection nonetheless, we provide appropriate loss-tolerant data stream operators.

Flexibility: There is no established protocol stack for sensor networks – a large variety of radio configurations, MAC, routing, and application layer protocols are in use. To support this open protocol space, we provide a packet capturer that works with a large variety of MAC protocols and radio configurations, as well as a flexible packet parser.

Reliability: The DSN should provide reliable wireless communication. We use Bluetooth for this purpose, which has been designed as a cable replacement, employing frequency hopping and other techniques to minimize loss.

In the first part of this paper we present a concrete instance of the above approach called SNIF (Sensor Network Inspection Framework) which is – as the name suggests – intended as a widely applicable framework for passive inspection. The second part of the paper contains an extensive case study of how SNIF can be applied to so-called data gathering applications. In particular, our case study can detect similar problems as approaches for active inspection in [19], [22].

II. SNIF

SNIF is a general framework for passive inspection of multi-hop sensor networks to detect problems related to individual nodes (e.g., reboot, death), wireless links, paths (e.g., routing failures, loops), or global problems (e.g., partitions). SNIF consists of a deployment support network (DSN) that acts as a distributed network sniffer. Each of the DSN nodes implements the receiver part of the sensor network protocol stack, namely receive-only *physical layer* and *media access*, as well as a *packet decoder* to extract the contents of overheard packets. All overheard packets are routed to the DSN sink, which executes a *data stream processor* to analyze packet streams for problems. The results of this analysis are displayed by a *user frontend*. Below we give an overview of these components. More details can be found in a technical report [21].

A. 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. 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., operate in different frequency bands). Also, the DSN radio should support the formation of a robust network with negligible message loss and high bandwidth. Since the data stream processor needs to examine temporal relationships between packets overheard by different DSN nodes, internal time synchronization of DSN nodes is necessary. The DSN is installed alongside the actual sensor network and may be removed as soon as deployment is finished and 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 [31], which provides two radio frontends: 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 [3] for details). A laptop computer with Bluetooth acts as the SNIF sink that connects to a nearby DSN node. This DSN node thereupon forms the root of an overlay tree spanning the whole DSN and 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 peforms 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. Bluetooth provides an accuracy of 1.25 milliseconds per hop.

One might argue that the deployment of the DSN may be as difficult and error-prone as deploying the sensor network itself. However, as the lifetime of the DSN is short (in the order of days), energy and resource constraints are not a primary issue here. This enables us to use more reliable networking technologies such as Bluetooth. In fact, Bluetooth has been designed as a cable replacement and employs techniques such as frequency hopping and forward error correction to provide highly reliable data transmission.

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.

Our generic PHY implementation supports configurable carrier frequency, baud rate, and checksumming details. We assume that the sensor network uses a single frequency for communication (which is the case with current implementations) such that a single-channel radio is sufficient to overhear WSN traffic.

Regarding MAC, we expoit the fact that – regardless of the specific MAC protocol used – a radio packet always has to be preceded by a preamble and 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 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.

```
// PHY+MAC parameters
cc.freq  = 868000000;
cc.baud =19200;
cc.sop   = 0xcc33;
cc.crc   = 0x1021;
// encoding: endianness + alignment
encoding.endianness = " little ";
encoding.alignment  = 1;
// type  definitions  and constants
typedef  uint16_t    mote_id_t;
typedef  uint8_t      quality_t ;
struct   link_quality_t   {
      mote_id_t  id;
      quality_t   quality ;
};
const  int  LINKESTADV = 2;
default.packet = "TOS_Msg"; // default  packet  type
struct  TOS_Msg {
      uint16_t  addr;
      uint8_t  type,  group,  length;
      int8_t  data[length];  //  variable  payload  size
      uint16_t  crc;
};
struct  LinkAdv : TOS_Msg.data (type == LINKESTADV) {
      mote_id_t  id;
      struct   link_quality_t   links [];  //  variable  array  size
};
```

Fig. 1. A SNIF configuration file.

Fig. 1 shows an excerpt of a sample configuration file for inspecting a TinyOS application running on MICA2 motes. 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 0x33cc. The used CRC polynomial is 0x1021.

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 (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.

Consider Fig. 1 for an example, which describes link advertisement packets used by the Multihop routing service implemented in ESS [11]. Line 17 defines the struct `TOS_Msg` as the default packet layout. The LinkAdv PDU used by ESS, is encapsulated in the field `TOS_Msg.data`, but only if the `TOS_Msg.type` is equal to `LINKESTADV`. Arrays of variable size are supported, where the size is either contained in the packet (e.g., for `TOS_Msg.data`), or inferred from the packet size (e.g., for `LinkAdv.links`).

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 routed to the SNIF sink.

D. Data Stream Processor

The DSN outputs a stream of overheard packets that needs to be analyzed to detect any problems in the WSN. To enable an efficient deployment process, this analysis should be performed *online*, allowing an engineer to go out and study and fix affected nodes while the problem is still present.

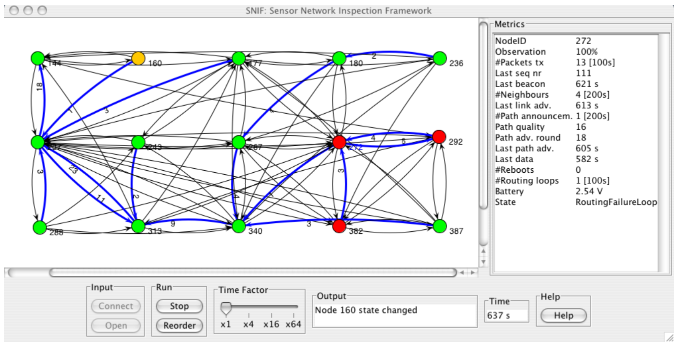
Given these preconditions, we decided for a data stream processor to perform online analysis of packet streams. Here, a data stream is an unbounded sequence of records. A data stream processor provides three basic abstractions: *sources* that produce data streams, *sinks* that consume data streams, and *operators* that modify data streams. 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. Mainly motivated by practical considerations (Java as implementation language, stability, open-source availability) we chose the PIPES data stream processor [5] for use with SNIF.

In SNIF, we model the DSN as a data stream source. An operator graph (being executed on the DSN sink) processes this data stream to detect indicators for problems, and sink nodes act as an interface to the user. A data stream record in SNIF is a list of attribute-value pairs with two special attributes holding record type and time stamp. The DSN produces records of type `Packet` with attributes holding the contents of an overheard packet. The syntax of the latter attribute names follows C syntax for accessing a field of a structure (e.g., `TOS_Msg.addr` to access the source adress of a packet in Fig. 1).

The data stream processor provides a number of basic operators to manipulate data streams, such as *Mapper* to rename record attributes, *Union* to merge multiple data streams into one where records are sorted by increasing time stamps, or *Filter* to drop records that do not match a given predicate. *TimeWindowAggregator* groups records according to a given attribute, removes duplicates, and computes aggregates over a time window. *ArrayIterator* provides access to array elements by creating N copies of each input record holding an array, where in the i-th output copy the array is replaced with element i of the array with size N.

Besides these generic operators, SNIF provides several data stream sources. The output of *DSNSource* consists of the packets overheard by the DSN, with records being sorted by increasing time stamp and duplicate packets (resulting from two or more DSN nodes overhearing the same sensor node) being removed. *EmSource* provides a similar interface to the EmStar [9] sensor network simulator, but is otherwise identical to *DSNSource*.

A typical application of SNIF is to infer the current state of inspected sensor nodes (e.g., node dead, node has no neighbors, etc.). To infer the state of a node, typically multiple data streams must be considered (e.g., a stream of periodic beacon packets to decide if a node is dead, a stream of neighborhood announcement packets to decide if a node has any neighbors). To this end, SNIF provides an operator *StateDetector* which groups records by type and node and stores the last record in each group. Whenever a group changes, an evaluation is



Metrics

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

Output: Node 160 state changed

Time: 637 s

Fig. 2. An instance of SNIF's user interface.

invoked to decide on the node state. We will refer to the above operators in Sect. III-D.

E. User Interface

To display any problems in the sensor network that have been detected by the data stream processor, SNIF provides a configurable user interface, which allows to display a real-time view of the network topology graph, where nodes and links can be annotated with application-specific information (e.g., state of a node, packet loss of a link) using a simple API. Also, logging and later replay of execution traces is supported. Fig. 2 shows an instance of this user interface for a typical data gathering application as discussed in the next section.

III. Case Study: Data Gathering Applications

Almost all existing non-trivial deployments are data gathering applications (e.g., [25], [28], [13]), where nodes send raw sensor readings at regular intervals along a spanning tree across multiple hops to a sink. In this case study we will therefore consider how SNIF can be applied to this application class. We first characterize the application in more detail and define the problems we want to detect. We then describe application-specific data stream operators to detect these problems and how they are used to form an operator graph. Finally, we evaluate the resulting inspection tool.

A. Application Model

Two prominent implementations of data gathering applications are the Extensible Sensing System (ESS) [11] using beacon-based multi-hop routing for data collection, and Surge using MintRoute [30] for data collection. Both implement a similar multi-hop tree routing scheme as described below. We will use ESS as an example throghout the paper, but our approach can be readily applied to other, similar implementations.

In ESS, all nodes broadcast *beacon messages* at regular intervals. 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. 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 ESS, beacons are sent every 10 seconds, path advertisements and link advertisements every 80 seconds, data message are generated every 30 seconds. 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.

B. Problems and Indicators

In [21] we studied existing deployments to identify common problems and passive indicators that allow to infer the existence of a problem from overheard network traffic. Below we summarize the problems that are considered in our case study and give passive indicators for their detection. Note the similarity to problems detected that can be detected tools for active inspection [19], [22].

Node death (fail stop): an affected node will not send any messages.

```
on receive beacon(src, seq, t):
  if (exists n[src]) {
    if (seq < n[src].seq) {
      if (n[src].seq < maxSeq − C)
        emit reboot(src, t);
      else if (t − n[src].t <
               (seq − n[src].seq) % maxSeq ∗ n[src].ival)
        emit reboot(src, t);
    }
    n[src].ival ← min (n[src].ival,
           (t − n[src].t) / (seq − n[src].seq));
  } else
    n[src].ival ← ∞;
  n[src].seq ← seq;
  n[src].t ← t;
```

Fig. 3. SeqReset operator.

```
on receive data(dst, seq, orig, t):
  if (exists p[seq|orig]) {
    if (p[seq|orig].dst = dst)
      emit retransmission(dst, seq, orig, t);
    src ← p[seq|orig].dst;
    p[seq|orig].dst ← dst;
  } else {
    src ← orig;
    p[seq|orig].dst ← dst;
  }
  emit data(src, dst, seq, orig, t);
```

Fig. 4. PacketTracer operator.

Node reboot: after reboot the sequence number contained in beacon messages will be reset.

Isolated node: the node is not listed as a neighbor in any link advertisement messages send by other nodes.

Node has no parent: the node fails to send path advertisement messages.

No path from node to sink: data messages sent by the node are not forwarded to the sink.

Node's path to sink loops: a data message originating from the node is sent twice to the same destination by different senders. Note that this is a special case of "no path from node to sink".

Node partitioned from sink: A node on the path from the node to the sink died and there is no alternate path available. Note that this is a special case of "no path from node to sink".

Although the above indicators are straighforward from a conceptual point of view, incomplete information makes their implementation less obvious as dicussed in the following section.

C. Application-specific Operators

This section presents application-specific operators that assist in detecting the problems described in Sect. III-B. The primary challenge here is to deal with incomplete information due to i) the DSN failing to overhear packets, and due to ii) information that would be needed to detect a problem not being expliciteIy included in messages.

SeqReset: This operator detects node reboots exploiting the fact that the sequence number containted in beacon messages will be reset after reboot. The main challenge here is to tell apart a wrap-around of the sequence number from reboot in case of lost beacon messages. The algorithm in Fig. 3 maintains a data structure n that holds for each node i the last sequence number *n[i].seq*, last time stamp *n[i].t*, and minimum interval *n[i].ival* between successive beacons. Whenever a beacon with source address *src*, sequence number *seq*, and time stamp t is received, the algorithm checks if *seq* is smaller than the last sequence number *n[src].seq* seen for this node. If the last sequence number is far apart from maximum sequence number *maxSeq* (parameter C must be selected such that loss of C consecutive beacon messages is highly unlikely), then *src* has rebooted. Otherwise, we apply an additional check to distinguish reboots from wrap-arounds with lost messages. In case of a wrap-around, the time between the last and current beacon messages *t - n[src].t* must be greater than or equal to the minimum beacon interval *n[src].ival* times the number of beacon messages that were lost plus one *(seq - n[src].seq) % maxSeq*.

PacketTracer: To reconstruct the multi-hop path of a message through the network, we need to know source and destinations addresses of each message. Unfortunately, data messages do not contain per-hop source addresses (as message receipt is not acknowledged). Also, messages not overheard by the DSN result in "gaps" in the multi-hop path. PacketTracer infers a source address for each packet, making sure that there are no gaps in the multi-hop path. The algorithm in Fig. 4 exploits the fact that each multi-hop message contains the address of the originator *orig*, a sequence number *seq*, and per-hop destination address *dst*. The operator maintains a data structure *p* that contains the last destination address *p[seq|orig].dst* for each multi-hop message uniquely identified by the concatenation *seq|orig* of sequence number and originator

```
on receive data (src, dst, t):
  if (dst ∈ n[src].desc) {
    emit routingloop (src, dst, t);
    remove dst from n[src].desc;
  }
  desc ← (src, t) ∪ n[src].desc;
  foreach (dn, dt) ∈ desc {
    if (dst = sink) {
      if (dn ∉ n[sink].desc)
        emit goodpath (dn, t);
      else if (dt > n[sink].desc[dn])
        emit goodpath (dn, t);
    }
    n[dst].desc ←
        n[dst].desc ∪ (dn, max (n[dst].desc[dn], nt));
  }
```

Fig. 5. PathAnalyzer operator.

address. If an entry for packet *seq|orig* doesn't exist yet, then the sender is set to the originator of the packet, otherwise the sender is set to the destination of the previous packet. If no messages are lost, then this approach obtains correct sender addresses. Otherwise, packets may span multiple hops, resulting in a multi-hop path without gaps. The following operators rely on this property. PacketTracer uses a timeout-based garbage collector to reclaim memory for past multi-hop packets (not shown).

PathAnalyzer: This operator checks if a node has a good path to the sink and also detects routing loops. Here, a good 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 is the sink, and the destination address of p_i equals the source address of p_{i+1}. The algorithm in Fig. 5 maintains a set *n[i].desc* of routing tree decendants for each node *i*, where each descendent is a pair *(j, tj)* of a node *j* and time stamp *tj*, meaning that *j* had a good path to *i* at time *tj* according to the above definition. When a data message with source address *src* (obtained by PacketTracer), destination address *dst*, and time stamp *t* is received, we first check if *dst* is among *src*'s descendants, which indicates a routing loop. Then we add *src* and all of *src*'s descendants to *dst*'s descendants, updating the time stamps accordingly. Whenever a new descendant is added to the sink or the time stamp of an existing descendent of the sink is incremented, this indicates a good path from this descendent to the sink.

```
on receive data (src, dst):
  n[dst].nb ← n[dst].nb ∪ src;
  reset timeout (dst, src);

on timeout (dst, src):
  remove src from n[dst].nb;

on receive nodestate (src, state):
  if (state = ''dead'') n[src].nb ← ∅;

periodically:
  DFS (n, sink);
  foreach unvisited node nn
    emit partitioned (nn);
```

Fig. 6. TopologyAnalyzer operator.

TopologyAnalyzer: This operator detects network partitions between a node and the sink caused by dead nodes in cases where PathAnalyzer doesn't find a good path to sink for this node. The algorithm in Fig. 6 maintains an approximate set of downstream neighbors *n[i].nb* for each node *i*. When a data packet with source address *src* and destination address *dst* is received, *src* is added to *dst*'s neighbors and a (user-defined) timeout is activated to remove this neighbor unless another packet with same *src* and *dst* is received before the timeout expires. TopologyAnalyzer is also subscribed to a data stream of records holding node states (see Sect. III-D for details). Whenever such a node state record is received indicating death of node *src*, the neighbor set of *src* is emptied. Periodically, TopologyAnalyzer performs a depth-first search on the graph given by *n[].nb* starting at the sink and marking all visited nodes. All nodes that have not been visited are reported as partitioned.

D. Operator Graph

Our inspection tool will compute the state of each node, which is either "node ok" or one of the problems described in Sect. III-B. In this section we outline the data stream operator graph that computes these states. Eventually, this graph will generate a record describing a node's current state whenever the state of the node changes.

The node state is derived using the binary decision tree depicted in Fig 7. The leaves of this tree represent possible states of a node. The decision tree is implemented using the StateDetector operator described in Sect. II-D. Each decision in the tree requires an operator graph that extracts the required information from the stream of observed packets.

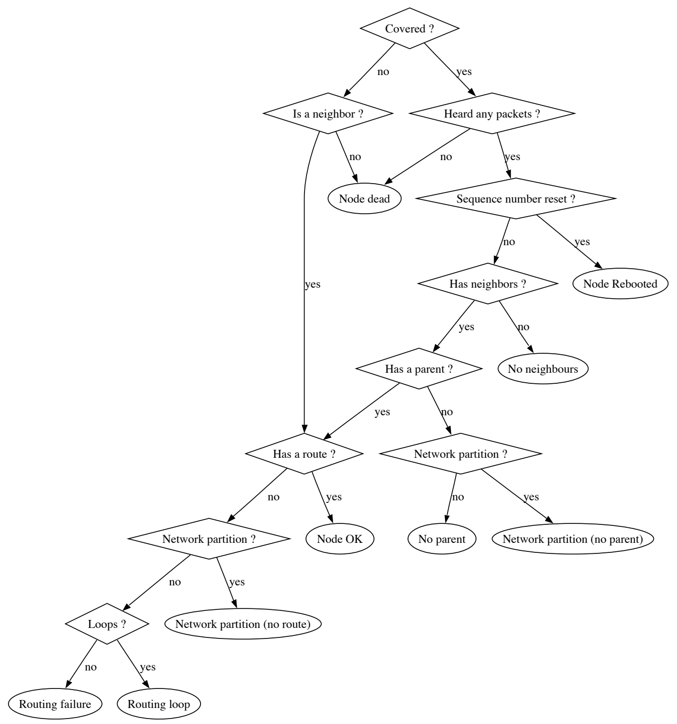


Fig. 7. Node state decision tree.

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 to compute the fraction of beacon messages that have been received. 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.

Sequence number reset?: This test succeeds is the node rebooted. To implement this test, DSNSource is filtered for beacon packets and SeqReset is applied to the resulting data stream.

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 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 twice 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.

In the above operator graphs, the time windows for TimeWindowAggregator are set to W times the interval between the packets they consider. For example, the time window in *Has a parent?* is set to $W \times$ 80 seconds, since path advertisement messages are considered which are sent every 80 seconds. That is, W is a global parameter and we will study its performance impact in Sect. III-E.

The structure of the decision tree is motivated by the desire to find and report the *root cause* of a failure. For example, a dead node (root cause) also has a routing problem (consecutive fault). Here, we want node death to be reported, but not the routing problem. Hence, in the decision three the checks to detect node death are located above the checks to detect a routing problem.

In addition to the above operator graph, we introduce several data stream sinks (not shown) to display relevant information in the graphical user interface as shown in Fig. 2. For example, 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.

E. Evaluation

To evaluate our case study, we used the same experimental setup as described in [19], where the Extensible Sensing System (ESS) [11] is executed in the EmStar emulator [9]. The reason for chosing EmStar instead of the real DSN as a data source for evaluation is the ease of injecting failures in a reproducible way with EmStar.

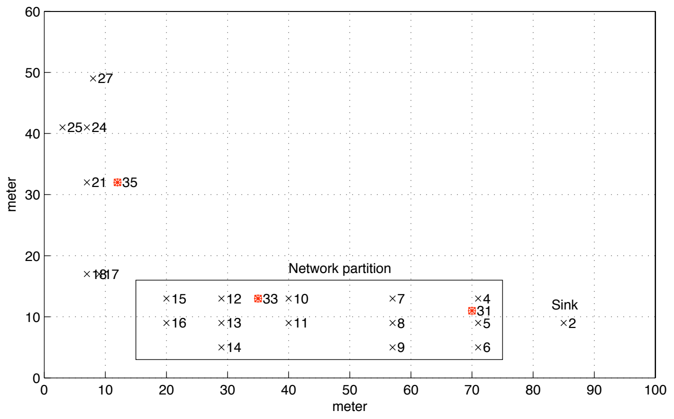


Fig. 8. Experiment setup: WSN (2-27) and DSN (31-35).

As depicted in Fig. 8, 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. 8). The link dump files of the DSN nodes generated by EmStar were used as input to the inspection tool. 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).

1) Accuracy and Latency: We study the accuracy (number and type of false error reports) and latency (time between failure injection and report) of our inspection tool. These metrics mainly depend on two parameters: the size of time windows used in the operator graph (i.e., the value of the time window factor W) and the amount of packet loss (i.e., fraction of sensor network messages that were not overheard by DSN nodes).

As most decisions regarding node state are based on packets received during a fixed time window, increasing W should improve accuracy (as operators then have more packets to base their decision on) and increase latency linearly (as more packets need to be collected before a decision is made). Increasing packet loss should degrade accuracy (as operators with fixed time windows then have less packets to base their decision on) and decrease latency (e.g., since node death is reported when no

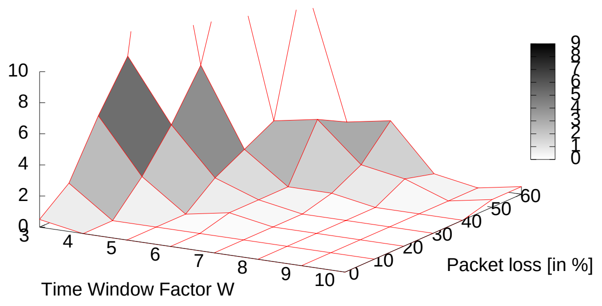
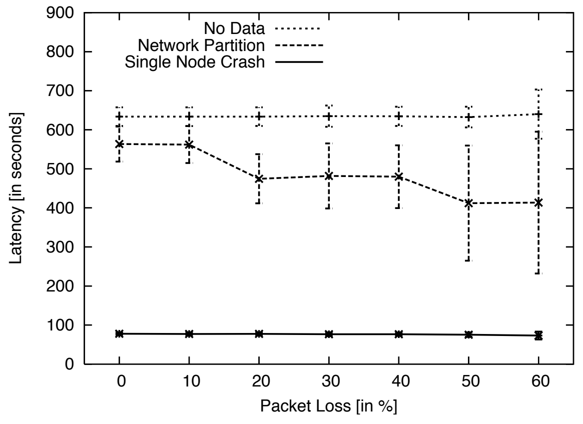


Fig. 9. Number of false reports as a function of packet loss and time window factor W.



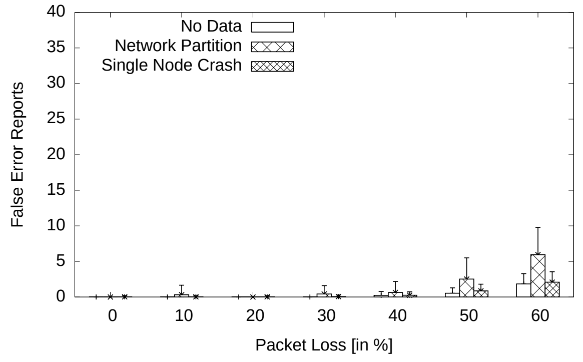


Fig. 10. Reporting latency and number of false reports as a function of packet loss for W=8.

packets are received from a node during a time window, loss of the last packets sent by a node before death will decrease latency).

In general, the latency to detect a problem is determined by the path of decisions leading to this problem in the binary decision tree depicted in Fig. 7. For example, the decision *Network partiton?* leading to state *Network partition (no parent)* can only be made when the previos decision *Has a parent?* has been made with a result of *no*. That is, the latency for detecting a given problem is a function of the maximum latency of the decisions in the decision tree on the path from the root to the leaf denoting this problem. In turn, the latency of a decision is determined by the size of time window(s) of the associated operator graph.

In order to assess the impact of W and packet loss on accuracy and latency, we ran a set of experiments injecting three types of faults into the network: node failure, network partition, and no data. The duration of each experiment was 30 minutes with faults being injected randomly between 10 and 15 minutes after experiment begin. In addition to the (small) packet loss of the DSN, we introduced additional packet loss by uniformly dropping a given fraction of the overheard packets. We report averages and standard deviation over multiple runs.

To guide the selection of W for a given amount of packet loss, we ran a first experiment without injecting any faults, varying both W and packet loss, counting the number of (false) error reports for each parameter choice. The averaged results over 10 runs are depicted in Fig. 9. The flat area of the graph shows feasible values for W given a certain packet loss. For a packet loss of 30% (a common value in single-hop sensor networks [25]), no errors were reported for $W \geq 7$, motivating our choice of $W = 8$ to study the impact of message loss in more detail as depicted in Fig. 10. Similarly, we chose a packet loss of 30% for a more detailed study of the impact of W as depicted in Fig. 11.

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 which is $W\times$ 10s with $W = 8$. As the beacons are sent every 10 seconds, we expect the latency to be between 70 and 80 seconds, which is confirmed by the experiments. Increasing packet loss does not have a significant impact on latency. The number of false positives is neglectable until 30% of packet loss and and raises significantly with more than

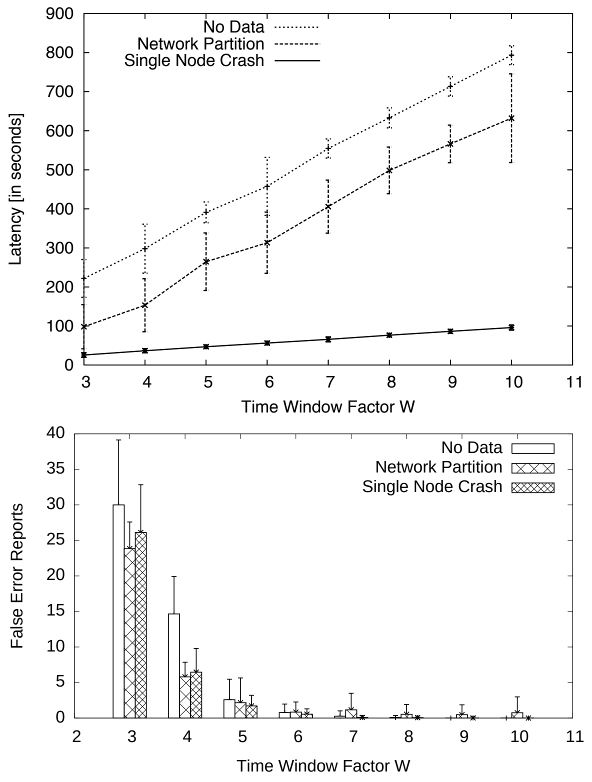


Fig. 11. Reporting latency and number of false reports as a function of time window factor W for 30% packet loss.

50% as depicted in Fig. 10 (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 (every 80s). For higher packet loss, we found *node dead* reports for working nodes. We never observed any false negatives. When varying W, we find (as expected) a linear increase of latency and an improvement of accuracy as depicted in Fig. 11.

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. We report the latency until the first node was classified as partitioned. As explained above, the latency of partition detection is bounded by the latencies of preceding decisions in the decision tree, namely *Has a parent?* and *Has a route?*, which both use a time window of $W\times$ 80 seconds. As *Has a route?* basically tracks multi-hop data packets which are sent often (every 30s), it reacts with shortly before 640 seconds. The *Has a parent?* fails, if no path announcements were observed during the time window. As explained above, increasing packet loss results in reduced detection latency.

In the third experiment, we injected faults into the Multihop routing component of single nodes such that an affected node stops sending data messages, while still broadcasting beacons and advertisements. 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 set to $W\times$ 80 seconds. As most nodes in the network forward packets for other nodes and data packets are sent every 30 seconds, the DSN should observe data packets until the fault is injected and the average latency should be close to the window size. The average of 633 $\pm$ 24 seconds for $W = 0$ and no packet loss confirms this. Again, in Fig. 11, the accuracy improves and latency increases linearly with W as expected.

2) 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.

3) A Bug in ESS: 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.

IV. Related Work

Complementary to SNIF is work on active debugging of sensor networks, notably Sympathy [19] and Memento [22]. Both systems require active instrumentation of sensor nodes and introduce monitoring protocols in-band with the actual sensor network traffic. Also, both tools support a fixed set of problems, while SNIF provides an extensible framework. Tools for sensor network management such as NUCLEUS [27] 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.

Also complementary to SNIF is work on simulators (e.g., SENS [24]), emulators (e.g., TOSSIM [14]), and testbeds (e.g., MoteLab [29]) 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. In [8], a deployment support network is used to provide a wireless backchannel to deployed sensor nodes. However, sensor nodes need to be physically wired to DSN nodes (requiring as many DSN nodes as there are sensor nodes) and sensor node software must be instrumented.

Passive observation by means of packet sniffing has also been applied to wireless (and wired) LANs [12]. 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. Very recently, two systems for passive analysis of WLANs have been proposed that use an approach similar to ours, namely WIT [15] and JIGSAW [6]. WIT follows an offline approach, merging redudant traces of network traffic collected by distributed sniffers. Using a detailed model of the 802.11 MAC, WIT then infers which packets have actually been received by the respective destination nodes and derives different network performance metrics. JIGSAW uses a similar approach to collect and merge traces, but then focuses on online inference of link-layer and transport-layer connections and their characteristics, also using a detailed model of the 802.11 MAC. In contrast, our approach is largely independet of the actual MAC used. Also, we focus on detecting a different set of problems as dicussed in Sect. III-B.

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], [2]) 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], [23]). However, we found it difficult if not impossible to express stateful SNIF operators using the SQL variants of these systems.

V. Conclusions

We presented a framework for passive inspection of deployed sensor networks, consisting of a distributed network sniffer, data stream processor, and user interface. The key advantage of this framework is that sensor networks need not be instrumented for inspection. The framework has been specifically designed to support different protocol stacks and operating systems. We showed how this framework can be applied to data gathering applications, demonstrating the our approach can detect typical problems encountered during deployment timely and accurately even in case of incomplete information. Using this tool, we found a bug in the ESS application. SNIF has been fully implemented and demonstrated at EWSN 2007 [20].

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP 2003*.

[2] P. T. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *ODSI 2004*.

[3] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-sensor networks. In *IPSN '05*.

[4] P. Buonadonna, D. Gay, J. M. Hellerstein, W. Hong, and S. Madden. Task: Sensor network in a box. In *EWSN 2005*.

[5] M. Cammert, C. Heinz, J. Krämer, A. Markowetz, and B. Seeger. Pipes: A multi-threaded publish-subscribe architecture for continuous queries over streaming data sources. Technical report, University of Marburg, 2003.

[6] Yu-Chung Cheng, John Bellardo, Péter Benkö, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *SIGCOMM 06*, Pisa, Italy, September 2006. ACM SIGCOMM.

[7] C. Cranor, T. Johnson, O. Spatcheck, and V. Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *SIGMOD 2003*.

[8] Matthias Dyer, Jan Beutel, Thomas Kalt, Patrice Oehen, Lothar Thiele, Kevin Martin, and Philipp Blum. Deployment support network - a toolkit for the development of wsns. In Koen Langendoen and Thiemo Voigt, editors, *4th European Conference on Wireless Sensor Networks (EWSN 2007)*. Springer, January 2007.

[9] L. Girod, J. Elson, A. Cerpa, T. Stathapopoulos, N. Ramananthan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *USENIX 2004*.

[10] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Sensys 2004*.

[11] 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.

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

[13] O. Visser K. Langendoen, A. Baggio. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *WPDRTS 2006*.

[14] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Sensys 2003*.

[15] R. Mahajan, Maya Rodrig, David Wetherall, and John Zahorjan. Analyzing the mac-level behavior of wireless networks. In *SIGCOMM 06*, Pisa, Italy, September 2006. ACM SIGCOMM.

[16] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02*.

[17] P. Padhy, K. Martinez, A. Riddoch, H. L. R. Ong, and J. K. Hart. Glacial environment monitoring using sensor networks, 2005.

[18] J. Polastre, R. Szewczyk, A. Mainwaring, D. Culler, and J. Anderson. Analysis of wireless sensor networks for habitat monitoring. In Cauligi S. Raghavendra, Krishna M. Sivalingam, and Taieb Znati, editors, *Wireless Sensor Networks*, chapter 18. Kluwer Academic Publishers, 2004.

[19] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys '05*.

[20] M. Ringwald, M. Cortesi, K. Römer, and A. Vialetti. Demo abstract: Passive inspection of deployed sensor networks with snif. In Koen Langendoen and Thiemo Voigt, editors, *4th European Conference on Wireless Sensor Networks (EWSN 2007)*, 2007.

[21] M. Ringwald, K. Römer, and A. Vialetti. Snif: Sensor network inspection framework. Technical Report 535, ETH Zurich, Zurich, Switzerland, 2006.

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

[23] M. Sullivan and A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX 1998*.

[24] S. Sundresh, W. Kim, and G. Agha. SENS: A Sensor, Environment and Network Simulator. In *Annual Simulation Symposium 2004*.

[25] R. Szewcyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *Sensys 2004*.

[26] 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'05*.

[27] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *EWSN 2005*.

[28] 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 '05*.

[29] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: a wireless sensor network testbed. In *IPSN 2005*.

[30] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Sensys 2003*.

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