Passive Distributed Assertions for Sensor Networks

Kay Römer
Institute for Pervasive Computing
ETH Zurich, Switzerland
roemer@inf.ethz.ch

Abstract— **When deployed in a real-world setting, many sensor networks fail to meet application requirements even though they have been tested in the lab prior to deployment. Hence, concepts and tools for inspection are needed to identify failure causes in situ on the deployment site. Tools for inspection should minimize the interference with the sensor network to, firstly, ensure that failures of the sensor network do not break the inspection mechanism, and, secondly, to ensure that the inspection mechanism does not change the behavior of the sensor network. In this paper, we propose passive distributed assertions (PDA) as a novel tool for identifying failure causes. PDA allow a programmer to assert certain predicates over distributed node states. Packet sniffing is used to detect failed assertions, thus minimizing the interference with the sensor network.**[1]

I. Introduction

When deployed in a real-world setting, many sensor networks fail to meet application requirements even though they have been tested in the lab prior to deployment [13]. For example, many sensor networks have been reported to deliver only a small fraction of the sensor data they are expected to produce [21], while others failed altogether [7]. The reasons for such failures can be manifold, including hardware problems (e.g., condensing humidity causing short circuits in a sensor), software bugs (e.g., timing problems that cause the microcontroller to reboot frequently), and networking problems (e.g., communicating nodes fail to wake up concurrently due to excessive clock drift caused by extreme temperature variations). Unfortunately, these problems are often not encountered during pre-deployment tests, because the environmental conditions that trigger these problems are hard to simulate in the lab. Therefore, sensor networks have to be inspected in situ on the deployment site to identify and locate failures and their causes.

[1]This invited paper is an extended version of an earlier paper [15] that appeared at REALWSN 2008. The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 5005-67322 (NCCR-MICS).

Resource limitations of sensor networks make in-situ inspection a hard problem. The strict need for energy efficiency results in sensor network designs that expose very little information about the system state to an outside observer: every bit of extra information a sensor node exposes about itself increases energy consumption. However, without sufficient information about the system state it is hard to identify failure causes. In practice, poor *visibility* [20] of the network state makes deployment of sensors networks a costly and cumbersome process.

Therefore, our goal is to increase the visibility of sensor networks in order to reduce the effort required to deploy sensor networks that do meet application requirements. Any such effort should, however, minimize the interference with the sensor network. In particular, one should minimize the probability that a (partial) failure of the sensor network also breaks the inspection mechanism. Also, the inspection mechanism should not change the behavior of the sensor network in significant ways. To this end, we recently proposed *passive inspection* [14], where "passive" refers to the fact that a sensor network does not have to be modified to allow inspection. In particular, we install a small number of additional nodes alongside the sensor network to overhear messages exchanged among sensor nodes. These "sniffer" nodes use a second, robust communication channel (e.g., a powerful radio operating on a different frequency or cable) to send the overheard messages to a base station, which generates a globally ordered message trace. As the lifetime of the sniffer network is relatively short, energy and resource constraints are not a major issue here.

By studying typical protocols used in sensor networks, we found that a great deal of information about the state of the sensor network can be inferred from a message trace. For example, we can detect node failures and node reboots without modifying the protocols used in the sensor network. We can even infer routing topologies or detect the existence of network partitions without touching the sensor network [14].

While passive inspection allows to infer basic network

state and failure symptoms without modifications of the sensor network, the actual failure *causes* often cannot be identified in a completely passive manner (e.g., *why* does a node fail to route data to the sink?). One of the reasons for this is that passive inspection cannot access the internal state of sensor nodes such as the values of certain program variables. Hence, we propose to apply *small* modifications to the software and protocols executing on sensor nodes to expose some information about the internal node state to the sniffer network to enable better identification of possible failure causes.

In particular, we propose a mechanism called *passive distributed assertions* (PDA), which allows a developer to assert certain properties of the distributed system state such as *there should be at least one cluster head among the neighbors of each node*. These PDA are inserted into the program code much like traditional assertions in C programs. To verify that a PDA holds, affected nodes would broadcast small amounts of additional information that can be overheard by a sniffer network. Analyzing the resulting message trace, one can detect failed distributed assertions, giving valuable hints about possible failure causes.

Using a sniffer network to check distributed assertions minimizes interference with the sensor network. Firstly, no complex distributed protocols are needed in the sensor networks to check assertions. Secondly, as long as a node can send messages, failures in the sensor network do not affect the ability to check distributed assertions.

We introduce passive distributed assertions in Sect. II, presentd the overall system architecture in Sect. III, and discuss important implementation aspects in Sect. IV. Related work is considered in Sect. V. Current status and directions for future work are discuss in Sect. VI.

II. Passive Distributed Assertions

Traditional assertions such as those implemented by `assert()` in the C language consist of a Boolean predicate over a snapshot of the program state. The latter consists of the values of some of the program variables at a certain point in the execution of the program. An assertion is said to have failed if the predicate evaluates to false. For example, in the assertion `assert(a == 10)`, the predicate is an equality test and the relevant system state is the variable `a`. The snapshot of the program state is the value of `a` during the execution of the assertion statement, assuming that assertions do not change the variables they reference. The assertion fails if the value of `a` does not equal 10. PDA extend this notion to a set of distributed nodes and their state variables.

A. Addressing Nodes

Similar to traditional assertions, a PDA is a Boolean expression over program variables. However, some of these variables may reside on remote nodes. These remote variables are prefixed by the address of the node. For example,

```
PDA(a == 100:a)
```

asserts that the value of the variable `a` on the node executing the PDA equals the value of the variable `a` on the node with address 100. However, in practice one often wants to specify relevant nodes by giving their abstract properties instead of their addresses. As in our initial example in Sect. I, we may want to assert that a node has at least one cluster head among its neighbors. Here, we would like to avoid digging out the addresses of all neighbors manually. Also, we want to operate on sets of nodes rather than individual nodes.

For this purpose, PDA offer node sets as a basic abstraction and special identifiers to specify common node sets such as `all` for all nodes in the network or `hood(N)` for all nodes at most N hops apart from the node executing the PDA. Here, `hood` refers to the neighbor table maintained by the networking stack.

To express distributed assertions over such sets of nodes, we offer the operator `count`. For example, the PDA

```
PDA(count(hood(1),
      $:is_clusterhead == 1) >= 1)
```

asserts that there is at least one neighbor of the node executing the PDA whose variable `is_clusterhead` equals one. `count` essentially iterates over all nodes in the set given as the first parameter, binding the special address $ to the current node and checking the predicate given as the second parameter. Two commonly used special cases of `count` are `any(S, p)`, which equals `count(S, p) >= 1` and `all(S, p)`, which equals `count(S, p) == |S|`. Further operators over sets of nodes are aggregation operations such as `min`, `max`, or `sum`. For example,

```
PDA(max(hood(1),
      $:is_clusterhead) == 1)
```

would also check that there is at least one clusterhead among the neighbors of the node executing the PDA, assuming that the variable `is_clusterhead` has a value of either zero or one.

B. Distributed Snapshots

Whenever a node executes a PDA statement, we need to obtain a distributed snapshot of all the relevant variables on all the nodes the PDA refers to. However, due to the passive nature of PDA, we cannot trigger a node from the outside to send a snapshot of its local state at a specific point in time. Rather, we offer the programmer a primitive to trigger sending node state whenever the value of a relevant state variable changes. In the above example, the programmer would have to insert the statement

```
SNAP(is_clusterhead)
```

immediately after each assignment to the variable `is_clusterhead`. Note that this could also be automated by applying a framework for analysis and transformation of source code such as [11].

Executing a SNAP statement results in the node broadcasting a message that contains the names and values of the variables specified as parameters to SNAP.

Likewise, each PDA statement triggers broadcasting the assertion predicate along with the current values of the local variables of the executing node that are referred to in the assertion.

C. Checking Assertions

To check for failed PDAs, the message trace – consisting of broadcast messages resulting from PDA and SNAP statements – obtained by the sniffer network is analyzed. We assume that each message contains an accurate timestamp of when the according SNAP or PDA statement was executed on the originating node. We discuss in Sect. IV how this can be achieved without synchronizing the sensor network.

From the SNAP messages, we can reconstruct the values of all relevant variables for each point in time. Hence, to check a PDA, we extract the timestamp of the PDA and lookup the values of the variables the PDA refers to at this point in time and evaluate the predicate of the PDA on these values. If the predicate does not hold, we notify the user, reporting the location of the PDA in the source code as well as the values of all variables the PDA refers to.

While the basic approach is simple, a fundamental issue is incomplete information resulting from failure of the sniffer network to overhear all messages. This cannot be avoided, as we cannot trigger sensor nodes to resend missing messages due to our passive approach. However, we can detect missing messages in the message trace by adding per-node sequence numbers to all messages. Thereby, we can detect missing messages for all but the last message from each node in the trace. When checking an assertion with timestamp t, for each remote variable referenced by the assertion, we lookup the latest SNAP message with a timestamp smaller than t and the earliest SNAP message with a timestamp larger than t in the trace. If the sequence numbers of these two messages differ by more than one for any referenced remote variable, we cannot decide the assertion and inform the user. Otherwise, we can decide the assertion and inform the user only if the assertion failed.

III. System Architecture

Figure 1 depicts the architecture of the proposed system for passive distributed assertions with its four main components: *preprocessing* of application source code which is then deployed on the *sensor network*, the *sniffer network* overhearing messages generated by PDA and SNAP statements, and the *assertion checker* evaluating the message trace to detect failed assertions.

Preprocessing of the application source code has two main purposes: firstly, reduction of the messages sizes of SNIF and PDA messages and, secondly, automatic insertion of SNAP statements into the source code. For reduction of message sizes, the preprocessor extracts all static information from PDA and SNAP statements. With static information we refer to all information which does not change across invocations of a single PDA or SNAP statement, such as the assertion expression, names and types of all referenced variables, as well as filename and line in the source code where a PDA statement occurs. Further, the preprocessor assigns a unique identifier to each PDA and SNAP statement by inserting the identifier as an additional parameter of the PDA or SNAP function call. The preprocessor also creates a so-called assertion table that maps unique identifiers to the static information of the associated PDA or SNAP statement. This assertion table is then passed to the assertion checker. This preprocessing step enables us to reduce the message sizes of PDA and SNAP messages significantly by transmitting only the unique identifier instead of all the static information. The assertion checker can then extract the identifier from a

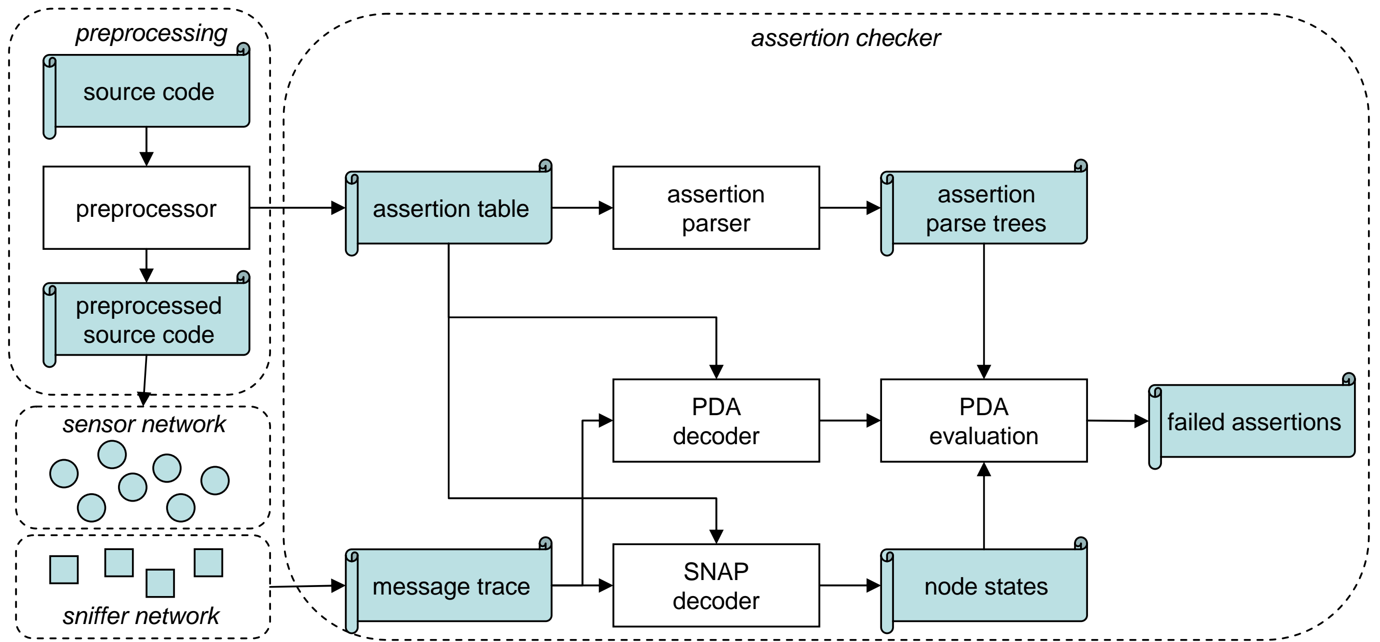

Fig. 1. System architecture for passive distributed assertions.

message and consult the assertion table to obtain the static information.

The second purpose of preprocessing is to automatically insert SNAP statements into the source code. For this, PDA statements would be analyzed to extract the referenced program variables. Wherever a value is assigned to such a variable in the source code, a SNAP statement would be inserted. However, automatic insertion of SNAP statements is subject of future work. In our current system, SNAP statements have to be inserted manually.

The preprocessed source code is then compiled and deployed on the sensor network, generating PDA and SNAP messages when executed. These messages are overheard by the sniffer network, producing a message trace as output. This message trace and the assertion table produced by the preprocessor form the input of the assertion checker that is executed on a portable computer that is connected to the sniffer network. Initially, the assertion checker parses the assertion expressions contained in the assertion table to validate the assertions and generate a parse tree for each assertion that can later be evaluated to check if an assertion holds. When processing the messages trace, SNAP and PDA messages are handled separately. Using the type information from the assertion table, SNAP messages are decoded and used to update a node state representation. The latter is essentially a lookup table which can be used to obtain the value of a certain variable on a certain node at a certain point in time.

PDA messages are decoded by using the type information stored in the assertion table. To check the distributed assertion, the according parse tree is evaluated, where references to remote variables in the parse tree are bound to values obtained from the node state representation. Note that the evaluation of a PDA must be delayed until appropriate snapshots (i.e., SNAP messages) have been received and processed for all remote variables referenced by the PDA. Hence, updates of the node state representation trigger the evaluation of a PDA. If an assertion fails or cannot be decided, an appropriate output message is generated and presented to the user. These output messages contain, among others, the assertion expression, the location of the assertion in the source code (file and line as stored in the assertion table), as well as values and timestamps of all referenced variables.

IV. Implementation

In this section, we discuss several important implementation aspects of passive distributed assertions, in particular the application programming interface exposed to the developer, the sniffer network, and aspects related time synchronization.

A. Language Mapping

Most sensor node platforms rely on C (or some dialect of C) as the system programming language. PDA and

SNAP statements introduced in the previous section can be mapped to C in a straight-forward way using a similar approach as for `printf()`. For example, the assertion:

```
PDA(all(hood(1), a == $:b))
```

where `a` and `b` are integer variables, would be mapped to the following C statement:

```
PDA("all(%g, %d == $:b)", hood(1), a);
```

Here, node sets such as `hood(1)` are treated like any other variable. `%g` acts as a placeholder for a node set. The function `hood(1)` returns a pointer to a data structure holding the set of neighbor addresses (i.e., a copy of the neighbor table maintained by the networking layer). For neighborhoods with more than one hop, the assertion checker will compute the N-hop neighborhood of a node by fusing the one-hop neighborhoods of relevant nodes. `%d` acts as a placeholder for an integer whose value is given in the variable argument list of the function call.

The `SNAP(b)` statement to emit a snapshot of variable `b` whenever its value changes would be mapped to:

```
SNAP("b=%d", b);
```

The implementations of the PDA and SNAP functions would simply scan the "format string" for placeholders starting with `%`, fetch the value from the variable argument list, and put this value into a message.

As discussed in Sect. III, a preprocessor extracts static information such as the format strings from the source code, assigns each PDA and SNAP statement a unique ID, and creates and assertion table that maps unique identifiers to static information, including the format strings. This way, only the unique ID is included in SNAP and PDA messages, such that the assertion checker can fetch the format strings from the assertion table to decode the messages.

B. Messages

To get a feeling for the size of messages produced by PDA and SNAP statements, let us consider the example PDA from the previous section. We need to include the node id of the sender (2 bytes), a sequence number to detect message loss (1 byte), unique number for the format string (2 bytes), a timestamp (4 bytes), and the value of variable `a` (2 bytes) – giving us a total of 11 bytes. It might even be possible to apply differential compression techniques (e.g., if the value of a variable changes only by a small increment) to reduce the size of messages. Further, it is possible to aggregate multiple PDA and/or SNAP messages into one to reduce overhead.

The messages resulting from PDA and SNAP statements don't have to be transmitted immediately. Rather, their transmission can be scheduled in a way that minimizes interference with the sensor network protocols. However, the best way to schedule these messages depends on the actual protocols used. For example, in time-triggered applications, where nodes send sensor readings at regular intervals, transmission of PDA and SNAP messages can be scheduled during the idle periods as other sensor nodes do not have to receive these messages. Another option would be to append PDA and SNAP message to other messages sent anyway by the sensor node to minimize the impact on medium access schedules and to eliminate preambles.

In any case, PDA and SNAP messages must be ignored by sensor nodes receiving them. Only sniffer nodes receive and process these messages.

C. Sniffer Network

Messages generated as a result of PDA and SNAP statements are collected by a sniffer network, consisting of additional nodes that are temporarily installed alongside the sensor network during the deployment process. The sniffer network can be removed as soon as the sensor network works as expected. Thus, the lifetime of the sniffer network is typically much shorter than than the lifetime of the sensor network. Hence, energy and resource constraints are not an issue with respect to the sniffer network.

Figure 2 (a) and (b) shows two different approaches to realize a sniffer network. In the figures, circles represent sensor nodes and squares represent sniffer nodes. Dashed lines depict wireless communication links. In (a), an online sniffer network is shown overhearing messages from the sensor network. Using a second (reliable and high-bandwidth) radio operating in a different frequency band than the sensor network radio, sniffer nodes forward collected messages to a base station (a laptop in the figure), which performs online analysis of the message trace. We have developed a working prototype [14] of such an online sniffer network based on the BTnode [1] platform, which offers a low-power ChipCon CC1000

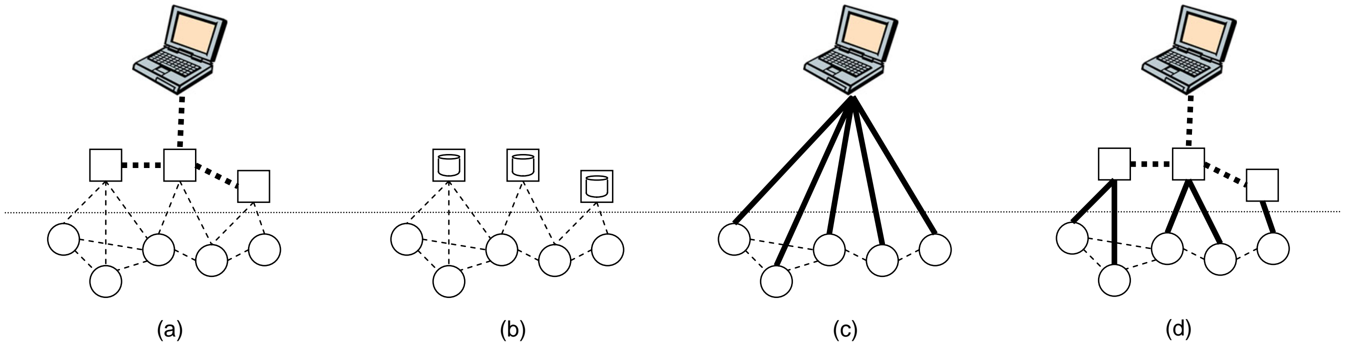

Fig. 2. (a) Online sniffer network (b) offline sniffer network (c) wired testbed (d) wireless testbed.

radio and a reliable and high-bandwidth Bluetooth radio. The ChipCon radio is used to overhear sensor network messages, while the Bluetooth radio is used to forward overheard messages to the base station performing trace analysis.

While an online sniffer network allows online detection of failed distributed assertions, it requires reliable networking of sniffer nodes. Figure 2 (b) shows an alternative solution, an offline sniffer network. Here, sniffer nodes do not forward overheard messages to a base station using a second radio, rather, these messages are stored in the flash memory (depicted by a small database symbol in the sniffer nodes). At some point, these nodes are collected to download and merge the individual messages traces from the flash memories. The merged trace is then analyzed offline to find failed assertions. Such offline sniffer networks have also been used in WLAN environments (e.g., [2]). In principle, messages could also be stored in the flash memories of the sensor node generating it, thus eliminating the need for additional sniffer nodes. However, this requires that the sensor network is accurately time-synchronized (see Sect. IV-D). Also, the sensor nodes themselves would have to be collected to download the traces.

In fact, distributed assertions can also be applied in testbed environments, where each sensor node has a wired back-channel to a base station as depicted in Fig. 2 (c). Here, the PDA and SNAP messages would be delivered over the wired channel rather than over the sensor network radio. A variant is a wireless testbed [3] as depicted in Fig. 2 (d). Here, additional nodes are deployed as for the online sniffer network. Sensor nodes have a wired back-channel to one of these additional nodes. Further, these additional nodes are connected to a base station using a reliable and high-bandwidth radio. While both wired and wireless testbeds have the advantage of using an out-of-band channel for collecting PDA and SNAP messages, their use is not very practical during deployment as both require to wire sensor nodes.

D. Time Synchronization

In Sect. II-C we relied on accurate timestamping of PDA and SNAP statements to check distributed assertions. However, due to the passive nature of distributed assertions, we cannot assume that the sensor network itself is time-synchronized. In this section we discuss how to obtain synchronized timestamps nonetheless.

The actual synchronization technique used depends on the type of sniffer network / testbed used. For online sniffer networks and wireless testbeds, we can assume that the sniffer nodes are accurately synchronized among each other. FTSP [10], for example, demonstrates that an accuracy of few micro seconds in a multi-hop network of 60 nodes is feasible using MAC-layer timestamping techniques. In the wired testbed, all messages are received by a single base station over wires, so no synchronization is necessary here.

In the offline sniffer network, sniffer nodes cannot communicate with each other, so the message traces collected by individual sniffer nodes need to be synchronized with each other in an offline fashion. In fact, this is possible if each message sent by the sensor network is overheard by multiple sniffer nodes. As such messages are broadcast messages, they will be received almost simultaneously by multiple sniffer nodes. Hence, these messages can be used as synchronization points to compute the time offsets between the sniffer nodes that received such as broadcast. In fact, this is equivalent to performing reference broadcast synchronization (RBS) [4] in an offline fashion. On motes, RBS has been reported to achieve an accuracy of few tens of micro seconds.

In summary, we can synchronize the sniffer / testbed nodes accurately using the approaches described above. What remains to be addressed is the fact that PDA and SNAP messages may be arbitrarily delayed before being sent as discussed in the Sect. IV-B. This is an issue because we need to timestamp the execution of PDA and SNAP statements on the sensor nodes.

Our approach to generate accurate timestamps for all PDA and SNAP statements is as follows. For each such statement, we read the local clock of the sending node twice: the first time when executing the PDA or SNAP statement resulting in timestamp t_1, and the second time using MAC-layer time stamping just before sending the first byte of the message to the radio or over the wire, resulting in timestamp t_2. We include $t_2 - t_1$ into the message.

The sniffer or testbed node receiving the message also reads its local clock when receiving the first byte of the message from the wire or from the radio using MAC-layer timestamping, resulting in timestamp t_3. As t_3 refers to the synchronized time of the sniffer network or testbed, $t_3 - (t_2 - t_1)$ is an accurate synchronized timestamp of the original PDA or SNAP statement execution. RITS [17] uses a similar approach, reporting an accuracy in the order of few tens of micro seconds.

Based on the above argumentation, we can expect an overall synchronization accuracy in the order of tens of micro seconds. The ATMEL ATMega 128, a typical microcontroller used on sensor nodes, runs at an average speed of 8 million instructions per second, which means that one instruction takes 1/8 μs on average. A synchronization error of 10 μs would then map to about 80 microcontroller instructions.

Note that it is possible that synchronization inaccuracy changes the result of the evaluation of a distributed assertion. For this to happen, the value of a remote variable referred to in the distributed assertion must be changed during a time window of 80 microcontroller instruction before or after execution of the assertion (assuming a synchronization error of 10 μs). Although the probability of this happening is non-zero, it is reasonably small given the small time window. Note that we can detect situations where this might be the case by checking if the timestamp of the PDA message is very close to the timestamp of one of the relevant SNAP messages. In this case, we inform the user of potentially incorrect decisions.

V. Related Work

In previous work, we proposed passive inspection with a sniffer network to inspect *unmodified* sensor networks [14]. While we can detect symptoms of many failures this way, we typically cannot identify the failure causes. Passive distributed assertions address this limitation by extending passive inspection to consider the internal node states.

In [20], visibility is introduced as an important metric for protocol design and visible protocols for data collection and dissemination are proposed. However, this approach focuses on the design of the networking protocols and does not expose internal node states.

Several systems have been proposed for debugging of sensor networks, notably Sympathy [12] and Memento [16]. Both systems introduce monitoring protocols in-band with the actual sensor network protocols. Also, both tools support a fixed set of problems, while passive distributed assertions are a generic mechanism. Tools for sensor network management such as NUCLEUS [19] provide read/write access to various parameters of a sensor node that may be helpful to identify failure causes. This approach also introduces protocols in-band with the actual sensor network protocols. Recently, the gdb source level debugger has been adopted to work on sensor nodes [23]. However, typical debugging operations such as single-stepping do significantly interfere with the sensor network, as the timing of operations is changed substantially. Also, the overhead of typical debugging operations is currently very high.

Complementary to PDA is work work on simulators (e.g., SENS [18]), emulators (e.g., TOSSIM [8]), and testbeds (e.g., MoteLab [22]). EmStar [5] 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. In [3], a wireless testbed is proposed, where sensor nodes still need to be physically wired to a testbed node, but testbed nodes communicate wirelessly with a base station. Hence, wireless testbeds may be suitable for use during deployment.

Packet sniffing has also been applied to wireless (and wired) LANs [6]. In particular, two interesting systems for passive analysis of WLANs have been proposed, namely WIT [9] and JIGSAW [2]. WIT follows an offline approach, merging redundant 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 destina-

tion 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. However, both systems do not provide access to internal node state.

VI. Conclusions

Deployment of sensor networks remains a major challenge: many sensor networks fail to meet application requirements even though they have been tested in the lab prior to deployment, necessitating concepts and tools to identify failure causes in situ on the deployment site. In this context, limited visibility of the network state to an outside observer is a key problem that needs to be addressed. To this end, we proposed passive distributed assertions, which allow developers to make assertions on the distributed system state while minimizing interference with the sensor network.

We are currently implementing the above proposal on the BTnode platform. Although we are confident that passive distributed assertions will be a valuable tool to help identify failure causes and make sensor networks more robust, many aspects need to be investigated in future work once the implementation is finished. Firstly, we need to study whether the proposed language for expressing assertions is sufficiently generic to address practical problems. Secondly, we need to study the traffic overhead introduced by PDA and SNAP messages and to which extent this traffic interferes with sensor network protocols. Thirdly, we need to study to which extent message loss and synchronization errors affect the frequency of undecidable assertions.

References

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

[2] Y.-C. Cheng, J. Bellardo, P. Benkö, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. In *SIGCOMM 2006*.

[3] M. Dyer, J. Beutel, T. Kalt, P. Oehen, L. Thiele, K. Martin, and P. Blum. Deployment Support Network - A Toolkit for the Development of WSNs. In *EWSN 2007*.

[4] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using re ference broadcasts. In *OSDI 2002*.

[5] 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*.

[6] 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 and Sons, 2006.

[7] K. Langendoen, A. Baggio, and O. Visser. Murphy Loves Potatoes: Experiences from a Pilot Sensor Network Deployment in Precision Agriculture. In *WPDRTS 2006*.

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

[9] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing the MAC-level Behavior of Wireless Networks. In *SIGCOMM 2006*.

[10] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi. The flooding time synchronization protocol. In *SenSys 2004*.

[11] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC 2002*.

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

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

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

[15] K. Römer and M. Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *REALWSN 2008*.

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

[17] J. Sallai, B. Kusy, Á. Lédeczi, and P. Dutta. On the scalability of routing integrated time synchronization. In *EWSN 2006*.

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

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

[20] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis. Visibility: a new metric for protocol design. In *SenSys 2007*.

[21] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI 2006*.

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

[23] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys 2007*.