PDA: Passive Distributed Assertions for Sensor Networks*

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

Junyan Ma
Institute for Pervasive Computing
ETH Zurich, Switzerland
Northwestern Polytechnical University, China
junyanma@inf.ethz.ch

ABSTRACT

Sensor networks are prone to failures and are hard to debug. This is particularely true for failures caused by incorrect interaction of multiple nodes. We propose a mechanism called passive distributed assertions (PDA) that allows developers to detect such failures and provides hints on possible causes. PDA allow a programmer to formulate assertions over distributed node states using a simple declarative language, causing the sensor network to emit information that can be passively collected (e.g., using packet sniffing) and evaluated to verify that assertions hold. This passive approach allows us to minimize the interference between the application and assertion verification. Further, our system provides mechanisms to deal with inaccurate traces that result from message loss and synchronization inaccuracies. We implement PDA on the BTnode platform and evaluate it using an extensive case study.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Assertion Checkers*; C.2.4 [**Computer Communication Networks**]: Distributed Systems

General Terms

Reliability, Languages, Verification

Keywords

sensor networks, distributed assertions, packet sniffing

1. INTRODUCTION

Problems in deployed sensor networks are not only common [9], but very hard to debug. Debugging requires visibility of the system state, but limited resources make it hard or impossible to extract the full system state from the sensor network. Debuggers compete with the debugged application for the limited system resources, resulting in pronounced interference between the debugger and the debugged application (e.g., probe effects). Sensor networks are large and dynamic distributed systems, thus debugging is not a single-node problem, but has to consider the interaction of many nodes. Sensor networks are systems with many input variables under control of the physical environment, hence the behavior of a deployed system may differ substantially from a testbed environment – necessitating debugging in situ on the deployment site.

The contribution of this paper is a mechanism called *passive distributed assertions* (PDA) to simplify debugging of sensor networks, with special emphasis on failures that result from incorrect interaction of multiple nodes. PDA allow a programmer to formulate hypotheses about the distributed state of the sensor network and include these hypotheses into the program code. Whenever the control flow reaches such a hypothesis in the code, our system checks if the hypothesis holds and alerts the user if this is not the case. For example, PDA could be used to express the hypothesis that at a certain point in the program execution, a sensor node should have at least one network neighbor that is a cluster head.

A key point of our approach is that we do not introduce complex protocols into the sensor network to check such hypotheses, rather we *passively* collect a message trace from the network, such that hypothesis checking is performed on this trace outside of the sensor network. By this, we minimize the interference of PDA with the sensor network application. In other words, we minimize the chance that introducing PDA alters the behavior of the sensor network application and that a failure of the application affects the PDA mechanism. A variety of approaches can be used to collect such traces in the lab and in the field, providing different trade-offs between the overhead required to collect the trace and the level of interference with the sensor network appli-

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

IPSN'09, April 13-16, 2009, San Francisco, California, USA.
Copyright 2009 ACM 978-1-60558-371-6/09/04 ...$5.00.

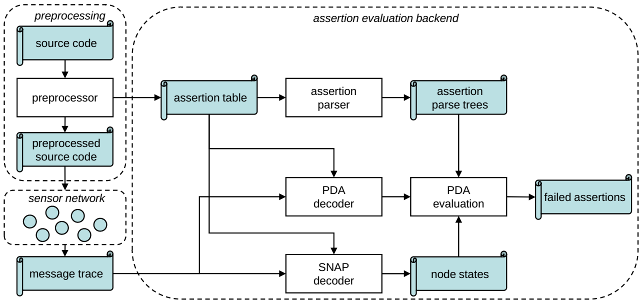


Figure 1: PDA system architecture.

cation.

A second important property of PDA is its ability to deal with inaccurate traces. For example, some messages may be missing from the trace or the correct ordering of the messages may not be established due to synchronization inaccuracies. Rather than producing wrong output in such cases (e.g., reporting a failed hypothesis where it actually did not fail), our system will flag such occurrences as *undecidable*.

In a recent position paper [13], we published the basic ideas underlying passive distributed assertions. The present paper turns these ideas into a complete system, presenting its detailed design, implementation, and evaluation. An overview of the system architecture can be found in Sect. 2. Sects. 3-8 present the detailed design and implementation aspects. An evaluation can be found in Sect. 9.

2. SYSTEM ARCHITECTURE

Our system allows a sensor node to publish part of is program state as so-called attributes, where an attribute has a name, a type (e.g., Boolean, Integer), and a value that may change over time as detailed in Sect. 3. Typically, an attribute maps directly to a program variable. A programmer can then insert distributed assertions over these attributes into the source code similar to the `assert` macro in the C programming language. A distributed assertion is essentially a Boolean expression over attributes as detailed in Sect. 4 below. However, these attributes may reside on different nodes. The semantics of such a distributed assertion is that whenever the control flow reaches a distributed assertion, the Boolean expression should evaluate to true. Otherwise, there is a mismatch of the programmer's hypothesis about the system state and the actual system state. In such a case, the user is informed, including details about the location of the assertion in the source code and the values of the referenced attributes. Distributed assertions can be disabled at runtime once the user is confident that the system works fine.

Fig. 1 depicts the architecture underlying our system. The basic idea is that sensor nodes execute a compact runtime system (Sect. 6) to publish small messages whenever an attribute value changes (so-called snapshot or SNAP messages) and when a distributed assertion is executed (so-called PDA messages). These messages can be either published in-band with other application messages on the radio channel, or out-of-band using a different channel (e.g., a serial cable attached to the node). A trace of these messages is collected and passed to the backend for evaluation (see Sect. 7). During trace collection, each attribute change (i.e., SNAP message) and execution of a distributed assertion (i.e., PDA message) is tagged with an accurate global timestamp. The backend uses this synchronized trace to evaluate assertions. For each PDA message in the trace, the backend reconstructs the values of all relevant attributes (using the SNAP messages in the trace) at the point in time the assertion was executed and evaluates the assertion on these attribute values. Details on the backend can be found in Sect. 8.

The source code of the application containing distributed assertions is run through a preprocessor (Sect. 5), which extracts information from the source code and modifies the source code with the goal of reducing the amount of information that has to be transmitted at runtime.

A key design decision behind this architecture is that each sensor node generates a trace of PDA and SNAP messages independent of all other nodes. Our system then *passively* listens to the sensor network to collect and evaluate these traces. This approach allows us to minimize the interference between assertion evaluation and the actual sensor network application. One the one hand, we thus minimize the probability that a partial failure of the sensor network application affects the ability to evaluate assertions. On the other hand, we thus minimize the probability that distributed assertions change the behavior of the application in significant ways. Different trace collection approach allow a user to carefully tune the trade-off between level of interference with the application and overhead for trace collection.

An alternative design would be to evaluate distributed assertions directly in the sensor network by introducing a protocol that allows a node to fetch values of attributes from remote nodes to evaluate a distributed assertion (e.g., as suggested in [16]). However, a failure in the multi-hop routing protocol would then also break assertion evaluation. Also, as distributed assertions may reference attributes on many distant nodes, the resulting traffic overhead may easily change the behavior of the sensor network application.

3. ATTRIBUTES

Attributes are an abstraction of the program state. Each attribute has a name, a type, and a value that may change over time. In many cases, attributes map directly to program variables, i.e., the attribute has the same name, type, and value as a program variable.

In order to enable the use of an attribute in a distributed assertion, a snapshot of the value of the attribute has to be published whenever its value changes. In our system, this is accomplished by the `SNAP(n,v)` function, where `n` is the name of the attribute given as a string and `v` is the value of the attribute. Consider the following example:

```
int a; ...; a=1; SNAP("a", a)
```

Here, a program variable of type integer is created and later assigned a value. Right after the assignment, the SNAP function is invoked to publish a snapshot of the new value of attribute a. It is also possible to take snapshots of multiple attributes with one invocation of the SNAP function as in

```
SNAP("a,b", a, b)
```

A special attribute is the neighborhood of a node, i.e., a set of node addresses identifying the direct neighbors of a node in the wireless network. Such a neighbor table is maintained by many MAC and routing protocols. As we will see below, this attribute can be very useful to formulate distributed assertions.

4. DISTRIBUTED ASSERTIONS

To formulate a distributed assertion, our system offers the function `PDA(o,e,c1,c2,...)` where `o` is a time offset, `e` is a Boolean expression over node attributes given as a string, and `c1,c2,...` are zero or more so-called evaluation constants. Consider the following example:

```
PDA(0, "a == 2:b")
```

This distributed assertion represents the hypothesis that the value of attribute a on the node executing the distributed assertion equals the value of attribute b on the node with address 2 at the point in time when the the control flow reaches the distributed assertion.

As illustrated by the above example, distributed assertions are specified by Boolean expressions over node attributes, using an extended syntax of expressions in the C programming language. Besides the usual arithmetic and Boolean operators, an extensible set of builtin functions is supported.

4.1 Delayed Assertions

Sometimes it is desirable to evaluate a distributed assertion at a point in time different from the point in time when the control flow reaches the assertion statement. For example, if a node sends a message to node 2 instructing it to set the value of its attribute b to the value of attribute a, one may want to check that node 2 has completed this operation after a certain amount of time. This can be accomplished by setting the first parameter of PDA to a value other than zero as in the following example:

```
PDA(100, "a == 2:b")
```

Here, the assertion will be evaluated 100 milliseconds after the control flow has reached the assertion statement.

4.2 Evaluation Constants

Often it is inconvenient to hardcode node addresses or other parameters as in the above example, where node address 2 cannot be changed at runtime. For this purpose, we introduce evaluation constants as in the following example:

```
PDA(100, "a == %addr:b", address)
```

Here, the placeholder `%addr` will be replaced with the value of the variable `address` before the assertion is evaluated[1]. Note the difference between evaluation constants and attributes. Attributes are published so that all nodes can use their values in distributed assertions, while evaluation constants are not published to other nodes.

4.3 Node Sets

In practice it is often necessary to formulate distributed assertions over sets of different nodes. For example, one may want to check that at least one node or all nodes out of a given set have a certain property. Such node sets can either be given as an explicit list of node addresses or by using builtin functions such as `nodes()` that returns the set of all nodes in the network or `hood(n)` which returns the set of nodes at most n hops apart from the node executing a distributed assertion. Note that these builtin functions are not evaluated on the sensor nodes, but only in the backend. The use of `hood(n)` requires that nodes publish their neighborhood as an attribute as described in Sect. 3. Knowing the individual neighborhoods (i.e., `hood(1)`) of all nodes, the backend can compute `hood(n)` for any value of n by computing shortest paths between pairs of nodes.

Special operators exist to formulate distributed assertions over node sets. These have the form `op(set,exp)`, where `op` is the name of the operator, `set` is a node set, and `exp` is an expression over node attributes that is evaluated for each node in the set. In `exp`, the special address prefix $ can be used as a placeholder for the currently considered node. Some example are:

```
PDA(0, "exists(hood(1), a == $:b)")
PDA(0, "all(hood(1), a == $:b)")
PDA(0, "count(hood(1), a == $:b) > 2")
PDA(0, "max(hood(1), $:b) > a")
```

From top to bottom, these assertions ensure that attribute b of at least one neighbor equals attribute a; that attribute b of all neighbors equals attribute a; that attribute b of at least two neighbors equals attribute a; that the maximum value of attribute b among the neighbors is greater than attribute a.

Real-world examples of distributed assertions can be found in Sect. 9.

5. PREPROCESSOR

The source code of the application, including PDA and SNAP statements as described earlier, is run through a preprocessor. The preprocessor performs type inference and type checking, and instruments the code to compresses assertions for lower communication overhead.

Both PDA and SNAP statements contain static informa-

[1] Note the similarity to the `printf` function in C. In fact, the PDA and SNAP functions are implemented in C as functions with a variable argument list.

tion that does not change over time, but is required by the backend to evaluate assertions. For example, the Boolean expression of assertions as well as their location in the source code (file name, line number) does not change over time. To avoid transmitting this information repeatedly as part of SNAP and PDA messages, the preprocessor extracts this static information, creates a table with one line for each PDA and SNAP statement and passes this table to the evaluation backend. Also, the preprocessor modifies the source code to include an index into this table (i.e., the row number) as an additional parameter into each invocation of the SNAP and PDA functions. This way, PDA and SNAP messages do only contain this index instead of repeatedly transmitting bulky static information.

6. NODE RUNTIME

The main purpose of the runtime system is to translate invocations of the PDA and SNAP functions into small messages. A trace of these messages is then collected (Sect. 7) and fed to the backend (Sect. 8), where the assertions are eventually evaluated. The runtime system consists of two main components, a message encoder that encodes the parameters of PDA and SNAP invocations into small messages, and a scheduler that schedules the transmission of these messages so as to minimize interference with the sensor network application.

6.1 Message Encoding

Three types of messages are created, SNAP messages containing snapshots of one or more changed attribute value, PDA messages representing the execution of a distributed assertion, and so-called periodic update (PU) messages. While PDA and SNAP messages are created as a result of invocations of the PDA and SNAP functions, PU are sent at regular intervals (in the order of tens of seconds). PU serve multiple purposes. Firstly, they enable the backend to detect node death (lack of messages from a node in the trace) and node reboot (sequence number contained in message resets to zero), which are both common problems that invalidate the node state. Secondly, PU messages allow the backend to decide that all SNAP messages containing attribute changes up to a certain point in time have been processed, assuming that the message trace contains messages in chronological order. Thirdly, PU contain copies of recently changed attributes that have already been reported through previous SNAP messages to compensate for lost SNAP messages.

All messages contain the address of the node that generated the message, a sequence number to allow the backend to detect missing messages in the trace, and two timestamps to support global synchronization. The first timestamp t_{exec} equals the point in time when the respective PDA or SNAP function was invoked or when the PU was triggered. The second timestamp t_{send} equals the point in time when the first bit of the message was sent and is obtained using MAC-layer timestamping. Note that we do not require that the clocks of the nodes are synchronized. However, using these unsynchronized timestamps we can compute a synchronized timestamp for each invocation of PDA, SNAP, and PU as described in Sect. 7.

In addition to this common header, PDA and SNAP messages contain the index that has been assigned by the preprocessor, allowing the backend to identify the static information belonging to the distributed assertion or snapshot encoded in the message. Finally, PDA messages contain the values of evaluation constants if any were given, while SNAP messages contain the values of changed attributes.

PU messages are similar to SNAP messages in that they contain the values of recently changed attributes. Moreover, a timestamp t_{latest} which equals the point in time of the latest value change among the recently changed attributes is also included to indicate that the values of all attributes in the message are valid between t_{latest} and t_{exec}.

6.2 Scheduling of Message Transmissions

If messages are sent in-band with the application traffic, then the transmissions of PDA, SNAP, and PU messages should be scheduled so as to minimize interference with the application protocols. A general strategy to reduce the overhead of distributed assertions is to transmit multiple messages back-to-back with a single, shared header. Our scheduler implementation follows this approach for snapshots of multiple attributes that are created with a single SNAP invocation, generating only a single SNAP message in this case. Further, our scheduler randomly delays the transmission of messages up to one second to reduce the chance of network congestion and collisions that may result when many nodes try to transmit PU or other messages simultaneously. If the radio supports this, a different frequency may be used to broadcast PDA and SNAP messages so as to avoid interference with application messages sent by other nodes.

Note that there is potential for further improvements in this area by exploiting application knowledge. For example, many data collection applications perform a sample, send, sleep cycle with long sleep periods. If packet sniffing is used to collect PDA, SNAP, and PU messages from the network (see Sect. 7), then these messages could be transmitted during the sleep phase of the application to avoid interference with the application. As part of future work we plan to investigate configuration mechanisms where a user can specify this type of application knowledge as an input to the scheduler.

7. TRACE COLLECTION

Trace collection is concerned with collecting PDA, SNAP, and PU messages from all sensor nodes, assigning a globally synchronized time stamp to each message in the trace, and merging all messages into a single chronologically ordered trace. Synchronization is needed as the evaluation of a

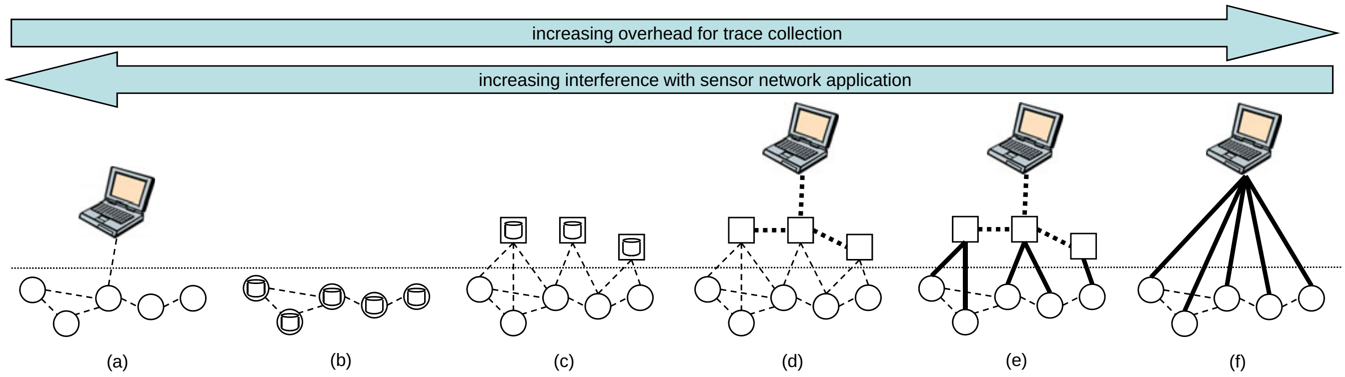

increasing overhead for trace collection
increasing interference with sensor network application
(a)
(b)
(c)
(d)
(e)
(f)

Figure 2: Trace collection: (a) In-band collection (b) Logging (c) Offline sniffer network (d) Online sniffer network (e) wireless testbed (f) wired testbed.

distributed assertion requires to compute the values of all attributes at the point in time when the assertion was executed on the node.

7.1 Trace Collection Approaches

The passive design of our system, where nodes generate traces of PDA, SNAP, and PU messages independent of each other, allows us to apply different approaches for collecting these traces as depicted in Fig. 2. Each such approach is characterized by a certain overhead (e.g., additional hardware, installation effort) and by a certain level of interference with the sensor network application. As illustrated in the figure, approaches with low interference typically require a high overhead, while approaches with low overhead result in higher interference. In particular, the amount of traffic that is transmitted in-band with the application traffic (i.e., over the same communication channel) determines the level of interference. The trace collection approaches also differ in the ability to perform an online evaluation of distributed assertions while the application is still executing.

The approaches depicted in Fig. 2 thus span a large design space, from which the application developer can chose the trace collection approach that best fits his specific requirements. During the lifetime of the application, different trace collection approaches can be used (e.g., a *wired testbed* in the lab and an *online sniffer* in the field) without affecting the ability to use distributed assertions.

With the *in-band collection* approach in Fig. 2 (a), PDA, SNAP, and PU messages are routed through the sensor network to a sink, where they are merged and fed to the evaluation backend. For example, many applications use a similar approach to collect sensor values from the network. Here, PDA, SNAP, and PU messages could be considered as "sensor values". This approach has the lowest overhead as no additional hardware is required, but results in high interference with the application as all traffic is transmitted in-band with application traffic through the whole network. This is an example for an online approach.

With the *logging* approach in Fig. 2 (b), PDA, SNAP, and PU messages are stored to a flash memory indicated by the small database symbols in the nodes. While this approach causes no interference with the application traffic, the nodes need to be collected to download the traces from their memories, resulting in substantial overhead. This is an example of an offline approach.

With the *offline sniffer* approach in Fig. 2 (c), sensor nodes broadcast PDA, SNAP, and PU messages in-band with the application traffic. However, other sensor nodes ignore these messages and do not forward them, resulting in significantly less in-band traffic compare to the *in-band collection* approach. Instead, an additional set of sniffer nodes (depicted as squares) is installed alongside the sensor network to overhear these messages and store them in their flash memories. Now, the sniffer nodes can be collected to download messages from their memories, while the sensor network remains operational. This approach has been used in [3].

With the *online sniffer* approach in Fig. 2 (d), sniffer nodes have a second, powerful radio that is free of interference with the sensor network radio (e.g., Bluetooth, WLAN). Using this second radio, sniffer nodes forward overheard PDA, SNAP, and PU messages to a sink where they are merged and fed to the backend for evaluation. In contrast to the *offline sniffer* approach, traces are processed online, but a reliable second radio channel is required. This approach has been used in [11] and is also applied in our case study in Sect. 9.

The *wireless testbed* approach in Fig. 2 (e) is similar to the online sniffer, but instead of sending PDA, SNAP, and PU messages in-band with application traffic, each sensor node is connected by wire (e.g., serial cable) to one of the sniffer nodes that forwards the messages over the second radio channel to the sink. This approach results in even less interference and message loss than *online sniffing*, but requires substantial overhead for wiring. This approach has been used in [4].

Finally, with the *wired testbed* approach in Fig. 2 (f), each node is wired to a sink. PDA, SNAP, and PU message are transmitted over the wire to a sink, where they are merged and fed to the evaluation backend in an online fashion. Many testbeds exist that support such a wired channel to each node, for example [18]. This approach is typically only feasible in the lab, while the other approaches could be applied both in the lab and in the field.

7.2 Trace Synchronization

The goal of trace synchronization is to assign a globally synchronized timestamp to each attribute change and to each execution of a distributed assertion. This is necessary to compute the values of all relevant attributes at the exact point in time when a distributed assertion is executed. The synchronization approach depends on the trace collection method used. We focus on *online sniffer network* and briefly outline synchronization approaches for other trace collection methods.

7.2.1 Online Sniffer Network

As we want to minimize interference of PDA with the sensor network, we do *not* require the sensor network to be synchronized. Instead, we synchronize the sniffer nodes among each other. In our implementation of the sniffer network, we use the synchronization algorithm described in [10], but any other synchronization algorithm could be used as well. A sniffer node now timestamps the reception of the first bit of each PDA, SNAP, or PU message using MAC-layer timestamping, obtaining a timestamp t_{recv} that refers to the synchronized time in the sniffer network. Now recall that each message contains timestamps t_{exec} when the PDA or SNAP statement was executed, and t_{send} when the first bit of the message was actually transmitted – both referring to the local, unsynchronized time of the originating sensor node. Assuming that the time for message propagation from sender to receiver is negligible, we can obtain a synchronized version of t_{exec} by computing $t_{\text{recv}} - (t_{\text{send}} - t_{\text{exec}})$. It has been shown that the accuracy of the resulting synchronized timestamp is in the order of 10 μs when applying such a time scale transformation [15]. It would also be possible to take into account the drifts of individual node clocks as in [12]. However, as $t_{\text{send}} - t_{\text{exec}}$ is usually small, the influence of clock drift is typically negligible.

Once a synchronized timestamp has been assigned to each PDA, SNAP, or PU operation in the trace in this way, the traces from all sniffer nodes are merged into a single trace, sorted in chronological order, and duplicate messages are removed. The resulting trace is then passed to the backend for assertion evaluation.

Although the above synchronization approach can provide an accuracy in the order of tens of μs, we have to take this inaccuracy into account when evaluating assertions, because an attribute value could change almost simultaneously with the execution of a distributed assertion. Here, it makes a difference if the timestamp of the assertion is a bit too early (uses old attribute value) or a bit too late (uses new attribute value). In the remainder of the paper we assume that there is a known upper bound Δ for the synchronization error. That is, a timestamp that has been obtained with the above approach does not differ from the correct timestamp by more than Δ time units.

7.2.2 Other Trace Collection Approaches

For *wireless and wired testbeds*, the same approach as for *online sniffer network* can be applied. In an *offline sniffer network*, the sniffer nodes cannot communicate with each other, hence it is not obvious how to synchronize them. However, if the sniffer nodes are deployed densely enough, then a single PDA, SNAP, or PU message is received by multiple sniffer nodes. As broadcast messages are received almost simultaneously by multiple receivers, a reception event can act as a reference point for synchronization. Hence, it is possible to perform reference broadcast synchronization (RBS) [5] in an offline fashion on the traces downloaded from the sniffer nodes, resulting in an accuracy in the order of 10 μs on typical sensor node hardware as demonstrated in [3].

Only for the *in-band collection* and *logging* approaches, the sensor network itself must be synchronized. Then, the t_{exec} timestamps contained in all messages are already synchronized (see Sect. 6) as they have been obtained from the synchronized node clocks.

8. ASSERTION EVALUATION BACKEND

The backend takes a trace of PDA, SNAP, and PU messages as input and evaluates the distributed assertions encoded as PDA messages in the trace, notifying the user of failed assertions and giving as much detail as possible on the failed assertion to assist the user in identifying the cause of the failure.

The backend consists of three main components, the state model which evaluates SNAP messages to reconstruct the values of node attributes over time, the assertion evaluator which evaluates an assertion using the state model to obtain the values of attributes at the time of assertion execution, and a user interface to display evaluation results.

8.1 Inaccurate Traces

The key challenge of assertion evaluation are inaccurate traces. Firstly, the input trace is likely to miss one or more messages depending on the trace collection approach. While traces obtained with *logging* and *wireless testbeds* are less likely to miss messages, approaches that involve wireless communication (in particular approaches involving passive sniffing due to the lack of acknowledgments and retransmissions) are likely to miss a substantial fraction of messages, often in the order of some percent. Secondly, as time synchronization is always inaccurate, the timestamps associated with attribute changes and distributed assertions are only estimates of the correct timestamps with a bounded error Δ

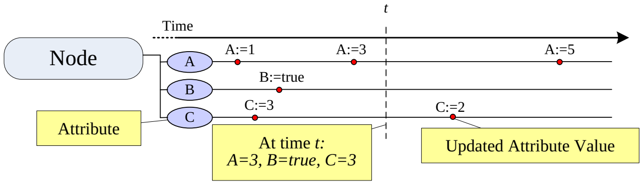


Figure 3: Reconstructing attribute values from snapshots.

(see Sect. 7).

As we cannot completely eliminate the causes of inaccuracies, assertion evaluation must be prepared to handle inaccurate traces. In particular, the evaluation backend should be able to identify assertions that cannot be evaluated correctly due to inaccuracies instead of reporting wrong evaluation results. Hence, in our system each attribute referenced by an assertion can be in one of tree possible states: *verified* (the value of the attribute at the time the assertion was executed is definitely known), *uncertain* (a possible value of the attribute at the time the assertion was executed is known, but may be incorrect), or *unknown* (a value for this attribute is not known at the time the assertion was executed). Likewise, an assertion has three possible outcomes: *success*, *fail*, and *unknown*. In addition, each assertion evaluation has four possible states: *tentative* (evaluation outcome may change if more messages are received), *finished* (evaluation outcome won't change any more and the assertion can be decided), or *undecidable* (evaluation outcome won't change any more and the assertion cannot not be decided).

8.2 State Model

The state model provides an interface to query the value and state of a given attribute of a given node at a given point in time. To provide this information, the network model processes the SNAP messages in the trace, which report the value changes of attributes in chronological order.

In an accurate trace (no lost messages, perfect synchronization), the value of an attribute at time t is given by the value reported in the SNAP message with the latest timestamp smaller than t. This is sufficient if the trace is processed offline, i.e., the backend sees the complete trace. However, if the trace is processed online, then we have to make sure that no later message with a timestamp smaller than t will arrive in the future. Hence, the value of the attribute remains *uncertain* until a message has been received with a timestamp greater than t. One of the purposes of PU messages is to ensure that such messages are regularly generated. As the trace is chronologically ordered, this will ensure that no messages with timestamps smaller than t will arrive in the future. Consider Fig. 3 for an example. Depicted is a sensor node publishing three attributes A, B, and C. Time progresses from left to right, small circles represent snapshots (i.e., value changes) of the attributes. At time t, the values of the attributes are $A = 3$, $B = \text{TRUE}$, $C = 3$. Note that all attributes are *verified* at time t, even attribute B, because at least one SNAP message with a timestamp greater t must have been received from the node to report snapshots for $A = 5$ and $C = 2$.

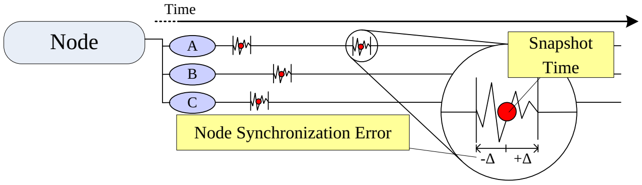


Figure 4: Impact of synchronization inaccuracy on attribute reconstruction.

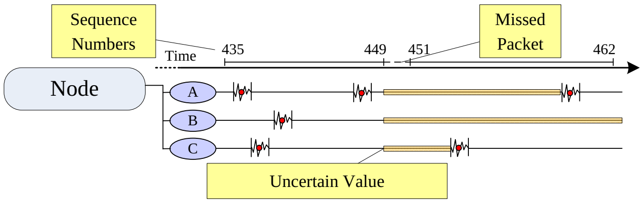


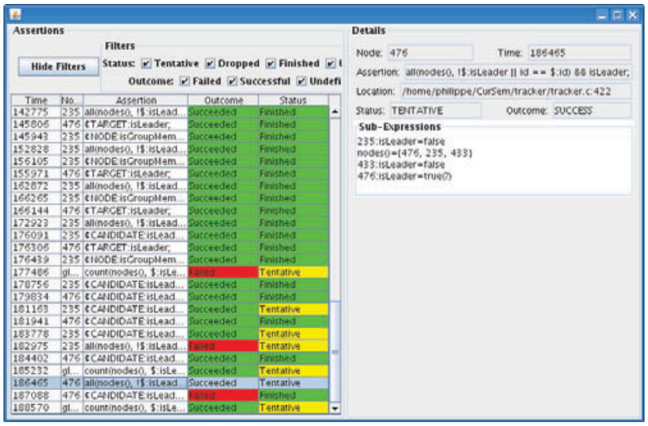
Figure 5: Impact of missing messages on attribute reconstruction.

In the presence of synchronization errors, things are slightly more complicated as depicted in Fig. 4, where each snapshot is enclosed by an interval $\pm\Delta$ that marks the maximum synchronization error. If the value of an attribute is requested during such an interval, the state of the attribute is *uncertain*, because it might be the case that in reality the attribute changed slightly before (after) the requested time t even if the timestamp suggests that the value changed after (before) t.

To detect lost messages, every message contains a monotonically increasing sequence number. If a certain sequence number is not present in the trace, then this indicates a lost message. In Fig. 5, the message with sequence number 450 is missing. Hence, all attributes of the node are marked *uncertain* starting with the timestamp of message 449, as attribute value changes reported in lost message 450 might have occurred arbitrarily close to the previous value change reported in message 449. An attribute remains *uncertain* until a later snapshot for this attribute has been received.

The network model also evaluates PU messages to detect node death and reboot. If for a certain amount of time no PU messages have been received from a node, then this node is assumed dead. The state of all attributes of this node is set to *unknown*.

When a node reboots, global variables are typically reinitialized. As the sequence number counter is also reset to zero upon reboot this way, a node reboot will manifest itself in a jump of the sequence number in messages to zero. In such a case, all attributes of the node are set to *unknown*.



Assertions

Filters

Status: ☑ Tentative ☑ Dropped ☑ Finished ☑ U

Outcome: ☑ Failed ☑ Successful ☑ Undefi

Time	No...	Assertion	Outcome	Status
142775	235	all(nodes(), !$:isLead...	Succeeded	Finished
145806	476	¢TARGET:isLeader;	Succeeded	Finished
145943	235	¢NODE:isGroupMem...	Succeeded	Finished
152828	235	all(nodes(), !$:isLead...	Succeeded	Finished
156105	235	¢NODE:isGroupMem...	Succeeded	Finished
155971	476	¢TARGET:isLeader;	Succeeded	Finished
162872	235	all(nodes(), !$:isLead...	Succeeded	Finished
166265	235	¢NODE:isGroupMem...	Succeeded	Finished
166144	476	¢TARGET:isLeader;	Succeeded	Finished
172923	235	all(nodes(), !$:isLead...	Succeeded	Finished
176091	235	¢CANDIDATE:isLead...	Succeeded	Finished
176306	476	¢TARGET:isLeader;	Succeeded	Finished
176439	235	¢NODE:isGroupMem...	Succeeded	Finished
177486	gl...	count(nodes(), $:isLe...	Failed	Tentative
178756	235	¢CANDIDATE:isLead...	Succeeded	Finished
179834	476	¢CANDIDATE:isLead...	Succeeded	Finished
181163	235	¢CANDIDATE:isLead...	Succeeded	Tentative
181941	476	¢CANDIDATE:isLead...	Succeeded	Finished
183778	235	¢CANDIDATE:isLead...	Succeeded	Tentative
182975	235	all(nodes(), !$:isLead...	Failed	Tentative
184402	476	¢CANDIDATE:isLead...	Succeeded	Finished
185232	gl...	count(nodes(), $:isLe...	Succeeded	Tentative
186465	476	all(nodes(), !$:isLead...	Succeeded	Tentative
187088	476	¢CANDIDATE:isLead...	Failed	Finished
188570	gl...	count(nodes(), $:isLe...	Succeeded	Tentative

Details

Node: 476 Time: 186465

Assertion: all(nodes(), !$:isLeader || id == $:id) && isLeader;

Location: /home/philippe/CurSem/tracker/tracker.c:422

Status: TENTATIVE Outcome: SUCCESS

Sub-Expressions

```
235:isLeader=false
nodes()={476, 235, 433}
433:isLeader=false
476:isLeader=true(?)
```

Figure 6: GUI of the assertion evaluation backend.

8.3 Assertion Evaluator

The assertion evaluator processes PDA messages contained in the trace. For each such message, the synchronized timestamp t is extracted and the values at time t of all attributes referenced by this assertion are requested from the state model. If all attributes are either *verified* or *uncertain*, then the Boolean expression of the assertion is evaluated on the attribute values and the result of the assertion is set to *success* or *fail* depending on the evaluation result. Otherwise, if at least one attribute is *unknown*, then the evaluation result is also *unknown*.

The state of the assertion evaluation is either *finished* if all attributes are *verified* or *undecidable* otherwise. If the trace is processed online, then it may take some time after the execution of the assertion until all nodes have sent respective SNAP or PU messages. During this time, the state of an assertion is *tentative* as the evaluation result may change later with the arrival of new messages. If an attribute changes its state from *unknown* or *uncertain* to *uncertain* or *verified* due to the arrival of a new message, all assertions referencing this attribute are reevaluated.

To evaluate operations over node sets (see Sect. 4), the set of nodes to evaluate the assertion on has to be computed. The set of all nodes, `nodes()` is known to the state model by virtue of the PU messages. To compute a neighborhood `hood(n)`, the value of the neighborhood attribute (see Sect. 3) of the node executing the assertion at time t is retrieved from the state model, which equals `hood(1)`. Then, the neighborhoods of all nodes in `hood(1)` at time t are retrieved and merged with `hood(1)` to obtain `hood(2)`. This process is repeated until `hood(n)` has been obtained.

The evaluation results are shown by the graphical user interface in Fig. 6. There is one column for each assertion execution, showing the evaluation result and state. By selecting a row, details are displayed on the right, including the location of the assertion in the source code and the values and states of all involved attributes. The displayed assertions can be filtered by various criteria.

9. CASE STUDY: TARGET TRACKING

We see passive distributed assertions as a tool to assist the developer throughout the whole development/test/deployment cycle of an application. That is, typically the programmer will add distributed assertions incrementally as he writes the code, rather than adding assertions to an existing application. Hence, to evaluate and gain experiences with our system, we implemented a complete nontrivial application from scratch on the BTnode [2] platform and used distributed assertions throughout the lifecycle of the application.

9.1 Tracking Application

The application is concerned with tracking the location of a single mobile target with a sensor network, similar to [1]. The target itself is also a sensor node that broadcasts beacon packets to allow its detection. These simplifications (e.g., single, cooperative target) are justified by the fact that we are mainly interested in passive distributed assertions and not in the application itself.

Sensor nodes are assumed to know their geographical location (x, y) in the plane and can take on different (also multiple) roles. Nodes that currently detect the target take on the *group member* role. One of the group members that is close to the location of the target additionally takes on the *leader* role and announces this fact to the group members. *Group members* regularly send a message to the leader containing their position. Using these messages, the leader computes the location of the target as the average of the locations of all *group members*. Nodes that do not detect the target but have a neighbor that is a *group member* are so-called *border nodes*. They overhear leader announcement messages sent by *group members*, enabling them to send reports to the leader when the target enters their range, turning them into *group members*. All other nodes are *idle*.

When the target moves out of the range of the *leader*, a new leader is elected among the *group members*. For this purpose, the *leader* also computes the direction of movement of the target using past target locations and computes the *group member* that lies furthest into the direction of movement in an attempt to minimize the number of handovers.

Greedy geographic routing is used to send messages between nodes across multiple hops. For this, the geographical location of the destination node must be known. The sender and intermediate nodes forward the message to the neighbor that lies closest to the destination. For neighbor discovery, nodes broadcast short messages containing their ID and location.

9.2 Distributed Assertions

During the development of the application, we placed a number of distributed assertions and snapshots in the code. In particular, all nodes publish two Boolean attributes *isLeader* and *isMember* that indicate if a node is a leader, a group member, both, or none. In the latter case, the node is either a border node if it recently received a leader announcement, or idle otherwise.

The following assertion is periodically executed by the leader code to check that the node executing the assertion is the one and only leader in the network. In other words, the node executing the assertion should have the *isLeader* attribute set and all other nodes (i.e., those with an ID that does not equal the ID of the node executing the assertion) must not have this attribute set.

```
A.1: PDA(isLeader &&
   all(nodes(), !$:isLeader || id == $:id)
```

Just before a node sends a target detection report to the leader, the following distributed assertion is inserted to check that the destination of the message (`%DEST` is an evaluation constant that is replaced with the actual address of the leader node) is actually the leader. Essentially, this is a consistency check that that a node's knowledge about the leader is consistent with the actual leader.

```
A.2: PDA(%DEST:isLeader)
```

An analogous distributed assertion is inserted into the leader code after the reception of a target detection report. Here, we check that the sender of a target detection report is actually a group member.

```
A.3: PDA(%SENDER:isMember)
```

The below assertion is executed when the current leader has lost the target and is about to send a handover request to the new leader candidate. This is a delayed assertion with a delay of 100 ms, checking that the new leader candidate has actually turned into the leader (i.e., has set its *isLeader* attribute) after 100 ms.

```
A.4: PDA(%CANDIDATE:isLeader)
```

In early versions of the code, we also exported the coordinates x and y of each node to verify the assumption underlying greedy geographic routing that there is indeed a node that is closer to the destination node (%DST) than the node executing the assertion. Here, `dist(x1, y1, x2, y2)` is a builtin function that computes the Euclidean distance between (x1,y1) and (x2,y2):

```
A.5: PDA(exists(hood(1),
         dist(x, y, %DST:x, %DST:y) >
         dist($:x, $:y, %DST:x, %DST:y))
```

9.3 Anecdotal Experience

Distributed assertions very tremendously helpful to debug early versions of the code, in particular of the leader election code. We encountered two classes of problems. Firstly, where a bug in the code resulted in a failed assertions, and secondly, where the code was actually correct, but the distributed assertion was wrong, i.e., the programmer's hypothesis about the system state as expressed by the assertion was wrong.

In experiments with the tracking application, assertion (A.4) failed occasionally depending on the mobility pattern of the target, meaning that the leader handover failed. By inspecting the values of attributes references by the assertion we found that the leader candidate was no longer a group member because he lost the target meanwhile but the leader didn't know. The failed assertions in Fig. 6 are an example of this behavior. Hence, we changed the code to try several different leader candidates until one accepts. Also, we changed the assertion to only require successful handover if the candidate is still a group member:

```
A.4': PDA(!%CANDIDATE:isMember ||
          %CANDIDATE:isLeader)
```

Similarly, assertion (A.3) occasionally failed depending on the mobility pattern of the target, meaning that the sender of a detection report is not a *group member*. It turned out that this was caused by the fact that the leader code that computes the target location is executed periodically and not immediately after receiving a new detection report from a *group member*. In the meantime, a former group member had lost the target. Here, the code was actually correct, but the assertion was misplaced. Hence, we moved it to the packet reception handler where it is executed immediately after receiving a detection report.

Finally, we observed a varying number of undecidable assertions depending on the type of assertion. While only a small fraction of the assertion executions (A.2-4) were undecidable, a larger fraction of assertion executions (A.1) couldn't be decided. The reason for this is that the assertion involves all nodes in the network, meaning that already a single lost message from any of these nodes could make the assertion undecidable.

9.4 Quantitative Results

To evaluate the overhead and accuracy of passive distributed assertions, we deployed an instance of the tracking application described above in our lab, consisting of 8 BTnodes arranged in a grid layout running the tracker application including distributed assertions and an additional BTnode that acts as the tracked target.

For trace collection, we used an online sniffer network developed in previous work [11], which is also based on BTnodes. Each BTnode includes a ChipCon CC1000 low-power radio and a Zeevo Bluetooth radio that operates in a different frequency band. The CC1000 is used to overhear messages exchanged in the sensor network. In addition, the sniffer nodes form a multi-hop Bluetooth network to route overheard messages to a laptop computer executing the backend software, which was implemented in Java. In our experiment, we use a sniffer network consisting of two nodes.

The experiment lasts for 300 seconds. At time 0, the network is switched on. At time 120 seconds, the target appears and remains static until the end of the experiment. We repeat the experiment five times and compute averages. The two

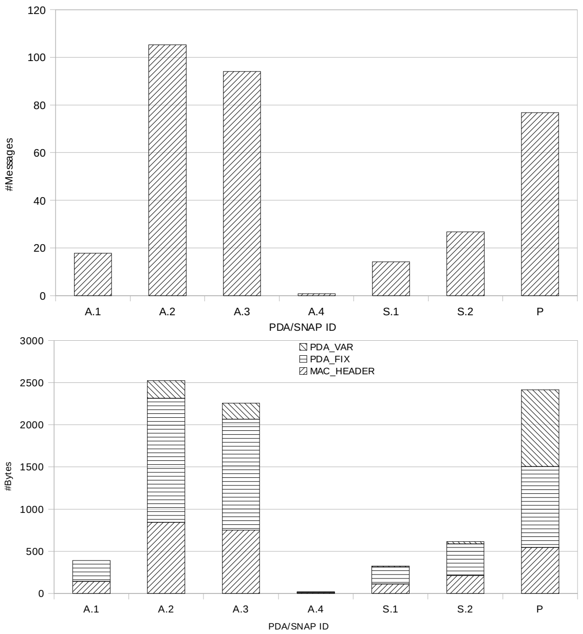


Figure 7: Number of messages and bytes sent due to distributed assertions.

main parameters of the distributed assertion system were set as follows: the PU interval was set to 30 seconds, while the time synchronization accuracy Δ was set to 1 millisecond.

Firstly, we consider the communication overhead caused by passive distributed assertions in terms of number of bytes and number of messages as depicted in Fig. 7, which shows summary values for the whole network, averaged over the experiment runs. We do not show standard deviations as these were very small. We break down the overhead to individual PDA, SNAP, and PU messages according to the following table:

ID	PDA/SNAP
A.1	PDA(isLeader && all(nodes(), !$:isLeader || id == $:id))
A.2	PDA(%TARGET:isLeader)
A.3	PDA(%SENDER:isMember)
A.4	PDA(%CANDIDATE:isLeader)
S.1	SNAP(isLeader)
S.2	SNAP(isMember)
P	Periodic Updates

The majority of the overhead was caused by periodic updates and assertions A.2 and A.3. The reason for this is that periodic updates are executed by all nodes, assertion A.2 is executed by all nodes that are group members, and assertion A.3 is executed by a leader when it receives a message from a group member. In contrast, assertions A.1 and A.4 are only executed by the single leader, A.1 in regular intervals and A.4 in case of a leader handover – which are rather infrequent events. In summary, about 336 messages containing 8551 Bytes were generated on average by the whole network, meaning that a single node generated about 42 messages or 1069 Bytes on average, which equals an average message rate of one message every 7.1 seconds or 3.6 Bytes per second. These values are quite acceptable given the typical communication bandwidth of sensor nodes. It must be noted, however, that this overhead depends on the specific application scenario (e.g., number, complexity, and execution frequency of snapshots and distributed assertions).

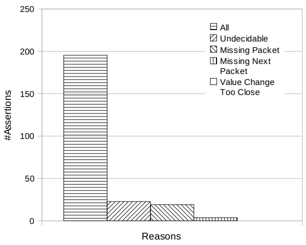


Figure 8: Undecidable assertions and their reasons.

Figure 7 (top) also shows a breakdown of the communication overhead into different components, namely the MAC headers (we used BMAC), fixed headers of PDA/SNAP/PU messages, and variable payload consisting of attribute values and evaluation constants. In our particular application, most of the overhead is caused by MAC headers and fixed headers. The MAC overhead could be reduced by combining multiple packets into one, which would, however, increase the amount of lost information, as the probability of message loss due to bit errors increases with message length, and as a single bit error causes all information in the combined packet to be lost. Combining multiple packets would, however, not reduce the overhead of fixed headers as these headers share little information across different messages. One possible approach would be some form of header compression. However, each of these headers in itself is quite small, requiring compression schemes with small initial overhead. Also, it is not advisable to apply differential compression to consecutive messages as then a single lost message would mean that the following messages cannot be decoded. We plan to investigate such compression techniques in future work.

Secondly, we investigate the accuracy of passive distributed assertions in terms of the fraction of assertions that cannot be decided as depicted in Fig. 8, which shows the total number of executed assertions and the number of undecidable assertions. In addition, we show the frequency of the different causes of undecidable assertions. *Missing packet* means that a SNAP message is missing in the trace that contains the value of an attribute that is relevant for the evaluation of an assertion. Note that even though periodic updates implement some form of retransmission of changed attribute values, attributes that change more frequently than the PU interval may still cause undecidable assertions. In fact, this is the main reason for undecidable assertions in our case. We also experimented with shorter PU intervals, but the increased network congestion resulted in higher packet loss with the used BMAC protocol and made things worse. Regarding *missing next packet*, recall from Sect. 8 that from each node that publishes an attribute that is referenced in an assertion, the backend needs to receive a message from this node with a timestamp later than that of the assertion to ensure that no relevant attribute changes are missing. *Missing next packet* means that such a message has not been received, which may occur at the end of the experiment, e.g., when the final PU message is lost. Finally, *value change too close* means that the value of an attribute at the time of execution of an assertion cannot be decided due to time synchronization inaccuracy (see Sect. 8). The latter did not occur during the experiments. Note that the different reasons are not additive, i.e., an undecidable assertion may have multiple causes if the assertion references multiple attributes.

The non-negligible fraction of undecidable assertions essentially means that our approach can miss transient or sporadic problems. Note, however, that for undecidable assertions the user can inspect the values and states of attributes referenced by an assertion to learn more about the system state.

The probability of undecidable assertions heavily depends on the trace accuracy, which in turn depends on the trace collection approach. In this respect, sniffing as used in this case study can be considered the worst case. Other approaches can either use acknowledged retransmissions (*in-band collection*), wired communication (trace readout with *logging*, *wired testbed*), or a combination of both (*wireless testbed*).

10. LIMITATIONS

10.1 Detectable Bugs

PDA supports the dectection of bugs that result from the incorrect interaction of multiple nodes. Both functional bugs (i.e., wrong action is performed) and timing bugs (i.e., right action is performed too early or too late) can be detected. As each distributed assertion considers the state of the system at a single point in time, it is not directly possible to reason about the evolution of the system state over time. Within these constraints, PDA offers a framework to detect arbitrary bugs, whereas other systems such as Sympathy [8] can only detect a fixed set of problems.

Another characteristic of PDA is that the developer specifies the correct system state explicitely by formulating an assertion, which may be difficult and can lead to situations where the assertion itself is incorrect. In contrast, other systems learn the correct system state automatically by observing the state during periods when the system is known to work correctly [6]. However, the latter approach is limited by the fact that correct system state that has not been observed during the learning phase will be considered an error later.

10.2 Overhead and Probe Effects

One of the goals behind PDA is to minimize the interference with the sensor network by minimizing the use of its resources. Nevertheless, PDA consumes resources (CPU cycles, memory, bandwidth) of the sensor network, the amount of which depends on the used approach for trace collection as discussed in Sect. 7.1. When using an online sniffer network, for example, this overhead is smaller than routing monitoring traffic through the sensor network to a sink as in Sympathy [8], but larger than completely passive approaches such as SNIF [11]. However, compared to the latter, PDA offers much better visibility of the node states.

The use of sensor network resources by PDA may result in probe effects, where bugs appear or disappear by introducing distributed assertions. For example, the use of distributed assertions may result in different memory layouts, such that bugs resulting from dangling pointers are subject to probe effects. Also, the timing of the program execution (computations, I/O, and communication) is altered by introducing distributed assertions, which may result in probe effects.

10.3 Scalability

There are three different scalability aspects with PDA. Firstly, the data rate of PDA/SNAP/PU traces generated by a sensor *network* increases linearly with the number of nodes in the sensor network. The trace collection infrastructure (e.g., online sniffer network) needs to be dimensioned such that it can handle the resulting aggregate bandwidth. Secondly, the data rate of a trace generated by a sensor *node* increases linearly with the frequency of PDA and SNAP executions, resulting in increasing interference with the application executing on the sensor node. Thus, PDA may not be appropriate to formulate assertions over attributes that change very frequently. Thirdly, the probability of an assertion being undecidable increases with the number of nodes an assertion refers to. This is true because an assertion is undecidable if the state of *at least one* of the nodes it refers to is unknown due to lost messages. Thus, PDA may not be appropriate to formulate assertions that refer to large numbers of nodes.

11. RELATED WORK

In previous work, we proposed passive inspection with an online sniffer network to inspect *unmodified* sensor networks [11]. Similarly, an offline sniffer network is used in LiveNet [3] to investigate the networking dynamics in an unmodified sensor network. While these approaches can detect symptoms of failures, the causes of failures typically cannot be identified in this way. Passive distributed assertions address this limitation by extending passive inspection to consider the internal node states.

We have published the basic ideas underlying our system in a position paper at a recent workshop [13]. In parallel to our own work, two other position papers have been published at recent workshops that suggest approaches that are similar to passive distributed assertions. Wringer [16] uses binary code rewriting to add watchpoints for global variables to the code that are triggered when the variable is assigned a new value. By introducing a Scheme interpreter into sensor nodes, node-local assertions can be formulated over variables. They also mention the possibility to formulate distributed assertions by collecting variable values from remote nodes using in-band routing protocols. However, they do not address distribution issues such as message loss and synchronization inaccuracies, causing incorrect evaluation results. Also, all communication is in-band, causing significant interference with the application. Further, they lack the flexibility of using different trace collection approaches as in our work. Finally, their approach requires writing Scheme code to formulate assertions (which may itself be prone to errors if the assertions are more complex), while we use a declarative language to formulate assertions. With MEGS [7], a programmer manually inserts code into the WSN application to send changes of variables over a side channel (e.g., wired testbed) to a central PC. At the PC, manually written Java code can be executed to check assertions over the distributed node states. As in Wringer, MEGS does not address issues such as message loss or synchronization inaccuracies. Also, formulating assertions requires writing Java code, which may itself be error-prone.

Several systems have been proposed for debugging of sensor networks, notably Sympathy [8] and Memento [14]. 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 [17] or Marionette [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 [20]. 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.

The concept of distributed assertions in itself is not new and has been used to debug distributed systems. We adopt this concept to sensor networks by introducing appropriate language abstractions (e.g., operations over neighborhoods), use passive trace collection to enable the developer to carefully balance overhead and interference, and provide mechanisms to deal with message loss and synchronization inaccuracies that are common in sensor networks. Also, in contrast to many systems that use logical time (e.g., Lamport or vactor time), our system is based on physical time as sensor networks are real-time systems (e.g., periodic activity is triggered by system timers). Finally, we provide a complete implementation on a sensor node platform with constrained resources.

12. CONCLUSIONS

We have presented passive distributed assertions, a tool to detect sensor network failures resulting from incorrect interaction of multiple nodes and to assist in identifying their causes. To verify distributed assertions, sensor nodes emit additional information that can be passively collected, such that the resulting trace can be evaluated outside of the sensor network, thus minimizing the interference between assertion verification and the application. A veriety of different trace collection approaches are supported that can be used in the lab and in the field, providing different tradeoffs between the amount of interference and the overhead required for trace collection. The system also detects and flags assertions that cannot be correctly evaluated due to incomplete traces or due to synchronization inaccuracies. We have applied the system in an extensive case study where it helped to fix several bugs and showed an acceptable overhead.

13. REFERENCES

[1] T. Abdelzaher and others. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS 2004*.

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

[3] B. Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *DCOSS 2008*.

[4] 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*.

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

[6] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys 2008*.

[7] M. Lodder, G. P. Halkes, and K. G. Langendoen. A global-state perspective on sensor network debugging. In *HotEmNets 2008*.

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

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

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

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

[12] K. Römer. Time Synchronization in Ad Hoc Networks. In *MobiHoc 2001*.

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

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

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

[16] A. Tavakoli, D. Culler, and S. Shenker. The case for predicate-oriented debugging of sensornets. In *HotEmNets 2008*.

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

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

[19] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN 2006*.

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