Deployment Techniques for Sensor Networks*

Jan Beutel[1], Kay Römer[2,3], Matthias Ringwald[2], Matthias Woehrle[1]

[1]Institute for Computer Engineering, ETH Zurich, 8092 Zurich, Switzerland
[2]Institute for Pervasive Computing, ETH Zurich, 8092 Zurich, Switzerland
[3]Insitute for Computer Engineering, University of Luebeck, 23538 Luebeck, Germany
Email: {beutel,woehrle}@tik.ee.ethz.ch, {roemer,mringwal}@inf.ethz.ch

Abstract. The prominent visions of wireless sensor networks that appeared about a decade ago have spurred enormous efforts in research and development of this new class of wireless networked embedded systems. Despite the significant effort made, successful deployments and real-world applications of sensor networks are still scarce, labor-intensive and oftenn cumbersome to achieve. In this article, we survey prominent examples of sensor network deployments, their interaction with the real world and pinpoint a number of potential causes for errors and common pitfalls. In the second half of this work, we present methods and tools to be used to pinpoint failures and understand root causes. These instrumentation techniques are specifically designed or adapted for the analysis of distributed networked embedded systems at the level of components, sensor nodes, and networks of nodes.

1 Introduction

Sensor networks offer the ability to monitor real-world phenomena in detail and at large scale by embedding wireless network of sensor nodes into the environment. Here, *deployment* is concerned with setting up an operational sensor network in a real-world environment. In many cases, deployment is a labor-intensive and cumbersome task as environmental influences trigger bugs or degrade performance in a way that has not been observed during pre-deployment testing in a lab. The reason for this is that the real world has a strong influence on the function of a sensor network by controlling the output of sensors, by influencing the existence and quality of wireless communication links, and by putting physical strain on sensor nodes. These influences can only be modeled to a very limited extent in simulators and lab testbeds.

Information on the typical problems encountered during deployment is rare. We can only speculate on the reason for this. On the one hand, a paper which only describes what happened during a deployment seldom constitutes novel research and might be hard to get published. On the other hand, people might tend to hide or ignore problems which are not directly related to their field of research. Additionally it is often hard to discriminate desired and non-desired functional effects at the different layers or levels of detail.

* The work presented in this paper was partially supported by the Swiss National Science Foundation under grant number 5005-67322 (NCCR-MICS), and by the European Commission under contract number FP7-2007-2-224053 (CONET).

In this chapter we review prominent wireless sensor network installations and problems encountered during their deployment. The insight into a sufficiently large number of deployments allows to identify common deployment problems, especially in the light of system architecture and components used, in order to gain a broader and deeper understanding of the systems and their peculiarities. In a second part we survey approaches and methods to overcome deployment problems at the level of components, sensor nodes, and networks of nodes. A special focus is on techniques for instrumentation and analysis of large distributed networked embedded systems at run-time.

2 Wireless Sensor Network Deployments

To understand the problems encountered during deployment, 14 different projects are reviewed with different goals, requirements and success in deploying the sensor network. The key figures for the projects surveyed are given in table 1. Main deployment characteristics are included such as the network size and the duration of a deployment. The column *yield* denotes the amount of data reported by the sensor network with respect to the expected optimum, e.g., based on the sample rate.

Deployment	Year	#nodes	Hardware	Duration	Yield	Multi-hop
GDI I	2002	43	Mica2Dot	123 days	16%	no
GDI II - patch A	2003	49	Mica2Dot	115 days	70%	no
GDI II - patch B	2003	98	Mica2Dot	115 days	28%	yes
Line in the sand	2003	90	Mica2	115 days	n/a	yes
Oceanography	2004	6	Custom HW	14 days	not reported	no
GlacsWeb	2004	8	Custom HW	365 days	not reported	no
SHM	2004	10	Mica2	2 days	up to 50%	yes
Pipenet	2004/2005	3	Intel Mote	425-553 days	$31\% - 63\%$	no
Redwoods	2005	33	Mica2Dot	44 days	49%	yes
Potatoes	2005	97	TNode	21 days	2%	yes
Volcano	2005	16	TMote Sky	19 days	68%	yes
Soil Ecology	2005	10	MicaZ	42 days	not reported	no
Luster	2006	10	MicaZ	42 days	not reported	no
Sensorscope	2006-2008	6-97	TinyNode	4-180 days	not reported	yes

Table 1. Characteristics of selected deployments.

In the above projects, a variety of sensor node hardware has been used as summarized in table 2. The majority of the early projects used the Mica2 mote [1] designed at the University of California at Berkeley and produced by Crossbow Technology Inc. or a variant of it (Mica2Dot, T-Node [2]). Its main components are an Atmel ATmega128L 8-bit microcontroller and a Chipcon CC1000 radio module for the 433/868/915 MHz ISM bands. More recent deployments often use the TI MSP 430 microcontroller due to its energy efficiency and more advanced radio modules such as the Chipcon CC2420 (implementing the 802.15.4 standard) on the Tmote Sky [3] or the Xemics XE1205 on the TinyNode [4] sensor nodes.

	Mica2(Dot)	T-Node	MicaZ	Tmote Sky
Microcontroller	ATmega128L	ATmega128L	ATmega128L	MSP 430
Architecture	8 bit	8 bit	8 bit	16 bit
Clock	7.328 MHz (4 MHz)	7.328 MHz	7.328 MHz	8 MHz
Program Memory	128 kB	128 kB	128 kB	48 kB
Data Memory	4 kB	4 kB	4 kB	10 kB
Storage Memory	512 kB	512 kB	512 kB	1024 kB
Radio	Chipcon CC1000	Chipcon CC1000	Chipcon CC2420	Chipcon CC2420
Frequency	433 / 915 MHz	868 MHz	2.4 GHz	2.4 GHz
Data Rate	19.2 kbps	19.2 kbps	250 kbps	250 kbps

	TinyNode	Intel Mote	Oceanography	GlacsWeb
Microcontroller	MSP 430	ARM7TDMI	PIC 18F452	PIC 16LF878
Architecture	16 bit	32-bit	8 bit	8 bit
Clock	8 MHz	12 MHz	<40 MHz	<20 MHz
Program Memory	48 kB	64kB	32 kB	16 kB
Data Memory	10kB	(*)	1536 B	368 B
Storage Memory	512kB	512kB	0.25 kB	64 kB
Radio	Xemics XE1205	Zeevo BT Radio	not specified	Xemics
Frequency	868 MHz	2.4 GHz	173 MHz	433 MHz
Data Rate	152.3 kbps	1 Mbps	10 kbps	9600 bps

Table 2. Sensor node characteristics. Data for Mica2Dot in parentheses. (*) The ARM7 core is a Von Neumann Architecture, hence there is no differentiation between data and program memory.

2.1 Great Duck Island

One of the earliest deployments of a larger sensor network was carried out in the summer of 2002 on Great Duck Island [1], located in the gulf of Maine, USA. The island is home to approximately 5000 pairs of Leach's Storm Petrels that nest in separate patches within three different habitat types. Seabird researchers are interested in questions regarding the usage pattern of nesting burrows with respect to the microclimate. As observation by humans would be both too costly and might disturb the birds, a sensor network of 43 nodes was deployed for 4 months just before the breeding season. The nodes had sensors for light, temperature, humidity, pressure, and infrared radiation and have been deployed in a single hop network. Each sensor node samples its sensors every 70 seconds and sends its readings to a solar-powered gateway. The gateway forwards the data to a central base station with a database and a two-way satellite connection to the Internet. During the 123 days of the experiment, 1.1 million readings have been recorded, which is about one sixth of the theoretical 6.6 million readings generated over this time.

In a book chapter [1], the authors analyze the network's behavior in detail. The most loss of data was caused by hardware-related issues. Several nodes stopped working due to water entering the sensor node casing. As all sensors were read out by a single analog-to-digital converter, a hardware failure of one of the sensors caused false readings of other sensors. Due to the transparent casing of the sensor nodes, direct sun light could heat the whole sensor node and thus lead to high temperature readings for nodes which are deployed above ground. Over time, various sensors report false readings such as humidity over 150% or below 0%, or too low or unreasonably high temperature. The temperature sensor of about half the nodes failed at the same time as the humidity sensor suggesting water inside the packaging. Although it did not directly cause packet loss as the gateway was always listening, several nodes did show a phase skew with respect to their 70 second sending interval. A crash of the database running on the base station resulted in the complete loss of data for two weeks.

After lessons learned from the first deployment, a second deployment was conducted in 2003 [5]. This time, two separate networks, a single-hop network of 49 nodes similar to the one in the first deployment and a multi-hop network with 98 nodes were deployed. The multi-hop network used the routing algorithm developed by Woo [6]. Again, the project suffered from several outages of the base station - this time caused by harsh weather. In the multi-hop network, early battery depletion was caused by overhearing in combination with low-power listening. In the pre-deployment calculation, the group did not account for an increased overhearing in the multi-hop network although it could have been predicted.

2.2 A Line in the Sand

An early project focussing on intrusion detection, classification and tracking of targets is ''A line in the sand'' [7]. The system consists of densely deployed sensor nodes with integrated binary detection sensors that collaboratively detect multiple objects.

The system implementation was evaluated by repeated deployments of ninety sensor nodes on three different sites in spring, summer, and fall of 2003 and smaller, focussed test deployments. The authors describe key problems encountered during the project. One of the main issues is the unreliability of the network. This is exacerbated by using a dense network of sensor nodes, which leads to network saturation and congestion and thus a considerable amount of collisions. Unexpected system interaction at scale diminished services for reliable communication. Additional unanticipated problems occurred due to hardware failures (e.g., debonding or desensitizing of sensors over time) and environmental conditions, exhausted batteries as well as problems due to incorrectly downloaded software. Unanticipated problems caused protocols to fail. The authors identify many faults that can be addressed by better packaging, hardware optimization and allowing for more redundancy, but indicate that many faults still require proper support for identification and resolution.

2.3 Oceanography

A small sensor network of 6 nodes was deployed in 2004 on a sandbank off the coast of Great Yarmouth, Great Britain [8] to study sedimentation and wave processes. A node did consist of a radio buoy for communication above the sea and a sensor box on the seabed connected by a wired serial connection. The sensor box had sensors for temperature, water pressure (which allows to derive wave height), water turbidity, and salinity. The authors reported problems with the sensor box due to last-minute software changes which led to cutting and re-fixing of the cable between buoy and sensor, and later, to the failure of one of the sensors caused by water leakage.

2.4 GlacsWeb

The GlacsWeb project [9] deployed a single-hop sensor network of 8 glacier probes in Norway. The aim of this system is to understand glacier dynamics in response to climate change. Each probe samples every four hours the following sensors: temperature, strain (due to stress of the ice), pressure (if immersed in water), orientation, resistivity (to detect whether the probe would be in sediment till, water or ice), and battery voltage. The probes were installed in up to 70 m deep holes located around a central hole which did hold the receiver of the base station.

In this deployment, initially, the base station only received data from seven probes and during the course of the experiment, communication with four probes failed over time. In the end, three probes were able to report their sensor readings. The base station experienced an outage. The authors speculate that the other probes might have failed for three reasons: Firstly, nodes might have been moved out of transmission range because of sub-glacial movement. Secondly, the node casing might have broken due to stress by the moving ice. And thirdly, clock drift and sleeping policy might have led to unsynchronized nodes which hinders communication.

2.5 Structural Health Monitoring

To assess the structural health of buildings, the Wisden [10] data acquisition system was conceived. Each node measures seismic motion by means of a three-axis accelerometer and forwards its data to a central station over a multi-hop network. The data samples are time stamped and aggregated in the network to compensate for the limited bandwidth. In the case of a seismic event, the complete data is buffered on a node for reliable but delayed end-to-end data transmission.

The authors report a software defect in their system, where an 8-bit counter was used for the number of locally buffered packets and an overrun would cause packets to not be delivered at all. Also, the accelerometer readings showed increased noise when the battery voltage did fall below a certain threshold.

2.6 Pipenet

Pipenet [11] is a sensor network to monitor pipeline infrastructures allowing for increased spatial and temporal resolution of operational data. It collects hydraulic and acoustic/vibrational data at high sampling rates and analyzes the data to identify and locate leaks in a pipeline. Pipenet uses a tiered architecture with the sensor network tier consisting of a cluster of battery-operated Intel Motes equipped with a data acquisition board. Sensor nodes are directly connected to the pipe, sense and process the collected information and directly send it via Bluetooth to the upper tier. First results of the initial trial from December 2004 to July 2005 are presented. One of the main problems encountered was battery exhaustion leading to long-term missing data. Short-term missing data is attributed to a watch-dog rebooting the gateway at midnight each day, leading to a loss of all messages in transit. Other sources for packet loss are environmental conditions such as rainfall and snow, where especially snow had a significant effect on packet loss. Hardware problems due to bad antennas and insufficient waterproofness attributed to additional problems in one of the clusters.

2.7 Redwood Trees

To monitor the microclimate of a 70-meter tall redwood tree, 33 sensor nodes have been deployed along a redwood tree roughly every two meters in height for 44 days in 2005 [12]. Each node measured and reported every 5 minutes air temperature, relative humidity, and solar radiation. The overall yield of this deployment has been 49%. In addition to the Great Duck Island hardware and software, the "Tiny Application Sensor Kit" (TASK) was used on the multi-hop network. The TinyDB [13] component included in TASK provides an SQL-like database interface to specify continuous queries over sensor data. In addition to forwarding the data over the network, each sensor node was instructed to record all sensor readings into an internal 512 kB flash chip.

Some nodes recorded abnormally high temperature readings above $40\,^{\circ}\mathrm{C}$ when other nodes reported temperatures between 5 and $25\,^{\circ}\mathrm{C}$. This allowed to single out nodes with incorrect readings. Wrong sensor readings have been highly correlated to low battery voltage similar to the report for the Structural Health Monitoring. This should have not been surprising as the used sensor nodes, Mica2Dot motes, did not employ a voltage converter and the battery voltage fell below the threshold for reliable operation over time. Also in this project, two weeks of data were lost due to a gateway outage. The data stored in the internal flash chip was complete but did not cover the whole deployment. Although it was estimated that it would suffice, initial tests, calibration, and a longer deployment than initially envisioned led to a full storage after about four weeks. In the end it turned out that the overall data yield of 49% was only possible by manually collecting all nodes and extracting the data from the flash memory on each node.

2.8 LOFAR-agro

A detailed report on deployment problems was aptly called "LOFAR-agro - Murphy Loves Potatoes" [2]. The LOFAR-agro project is aimed at precision agriculture. In summer 2005, after two field trials, 110 sensor nodes with sensors for temperature and relative humidity were deployed in a potato field just after potatoes have been planted. The field trials and the final deployment suffered from a long list of problems.

Similar to the oceanography project, an accidental commit to the source code revision control system led to a partially working MAC protocol being installed on the sensor nodes just before the second field trial. Later, update code stored in the nodes' external flash memory caused a continuous network code distribution which led to high network congestion, a low data rate and thus the depletion of all nodes' batteries within 4 days. The routing and the MAC component used different fixed size neighbor tables. In the dense deployment, where a node might have up to 30 neighbors, not all neighborhood information could be stored, which caused two types of faulty behavior. Firstly, the routing component of most nodes did not send packets to the gateway although the link would have been optimal. Secondly, as both components used different neighbor tables, packets got dropped by the MAC-layer when the next hop destination was not in its neighbor table. To allow nodes to recover from software crashes, a watchdog timer was used. Either due to actual program crashes or due to a malfunction of the watchdog handling, most nodes rebooted every two to six hours. This did not only cause data loss for the affected node but also led to network instability as the entries for rebooting nodes are removed by their neighbors from their neighborhood tables. As in other projects, the LOFAR-agro project suffered from gateway outages. In this case, a miscalculation of the power requirements for the solar-powered gateway caused a regular outage in the morning when the backup battery was depleted before the sun rises and the solar cells provided enough power again. The sensor nodes were programmed to also store their readings in the external flash memory, but due to a small bug, even this fallback failed and no data was recovered after the deployment. In total, the 97 nodes which ran for 3 weeks did deliver 2% of the measured data.

2.9 Volcanoes

In August 2005, a sensor network of 16 sensor nodes has been deployed on the volcano Reventador in Ecuador [14]. Each node samples seismic and acoustic data at 100 Hz. If a node detects a local seismic event, it notifies a base stations. If 30% of the nodes report an event in parallel, the complete data set of the last minute is fetched from all nodes in a reliable manner. Instead of immediately reporting all data, which would lead to massive network congestion and packet collisions with current low-power MAC protocols, the nodes are polled by the base station sequentially.

The first problem encountered was a software defect in the clock component which would occasionally report a bogus time. This led to a failure of the time synchronization mechanism. The team tried to reboot the network but this trigged another bug, which led to nodes continuously rebooting. After manual reprogramming of the nodes, the network was working quite reliably. A median *event yield* of 68 % was reported, which means that for detected events 68% of the data was received. As with other deployments, data was lost due to power outage at the base station. During the deployment, only a single node stopped reporting data and this was later confirmed to be due to a broken antenna.

2.10 Soil Ecology

To monitor the soil ecology in an urban forest environment, ten sensor nodes have been deployed near John Hopkins University in the autumn of 2005. The nodes have been equipped with manually calibrated temperature and soil moisture sensors and packed into a plastic waterproof casing. The sensor application was designed to store all sensor readings in the local flash memory which had to be read out every two weeks to guarantee 100% sensor data yield in combination with a reliable data transfer protocol.

However, due to an unexpected hardware behavior, a write to the flash memory could fail and an affected node would then stop recording data. Further parts of the data have been lost due to human errors while downloading the data to a laptop computer. Similar to previous deployments, the software on the nodes had to be updated and for this the waterproof cases had to be re-opened several times which led to water leakage in some cases.

2.11 Luster

A recent project for environmental monitoring is Luster [15], a system to monitor environmental phenomena such as temperature, humidity, or CO_2. LUSTER is an environmental sensor network specifically designed for high reliability. Its main goal is to measure sunlight in thickets in order to study the relation between light conditions and the phenomenon of shrub thickets overwhelming grasslands. Luster is a multi-layer, single-hop architecture replicated into multiple clusters that cover the entire deployment area: The lowest layer is a cluster of sensor nodes collecting and transmitting data at a configurable rate to a clusterhead gateway. The gateway nodes (Stargate microservers) form a delay tolerant network layer. For redundancy purposes an additional layer is introduced, the reliable distributed storage layer. This layer is made up of dedicated storage nodes overhearing data reported by sensor nodes at the lowest layer and passively storing it on SD cards, for physically retrieval at a later point in time. Overlapping sensor coverage increases reliability of the overall system.

The system was deployed at Hog Island. While two problems were caused by hardware failures, i.e., bad contacts to the sensor, and a non-responding storage node, the problems could be identified in the field using their inspection tool SeeDTV [16].

2.12 SensorScope

Sensorscope [17, 18] is a system for environmental monitoring with many deployments across Switzerland. Instead of few accurate and expensive sensors, many densely deployed inexpensive sensing stations are used to create accurate environment models. Instead of accuracy of individual sensors, the system design strives for generating models by high spatial density of inexpensive sensing stations. Each station is powered by a solar cell, which is mounted onto a flagstaff alongside the sensors and a TinyNode. Different environmental parameters are captured and gathered for later analysis. Most deployments are in the range of days to a couple of months. The environments are typically harsh, especially in the high mountain terrain deployments, e.g., on the Grand St. Bernard pass. Similar to the report of the LOFAR-agro project, the authors provide a detailed overview of issues and lessons learned in a comprehensive guide [18]. They again stress the importance of adequate packaging of the sensor nodes, and especially the connectors. With respect to these particularly harsh environments, substantial temperature variation showed considerable impact on the clock drift. However, while the clock drift typically affects the time synchronization of the network, in this case, it induced a loss of synchronization of the serial interface between the sink node and the GPRS modem. In an indoor test deployment, the authors report on packet loss due to interference where the interfering source could not be determined. Finally, a change of the query interval of the wind speed sensor, when moving from the lab to the field, caused counter overruns, which rendered a lot of sensor readings useless.

3 Deployment Problems

Based on the described problems typically found during deployments as surveyed in the previous section, a classification of problems is presented. Here, a "problem" is defined as a behavior of a set of nodes that is not compliant with a (informal) specification.

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

3.1 Node Problems

A common node problem is *node death* due to energy depletion either caused by "normal" battery discharge [7, 11], short circuits or excessive leakage due to inadequate or broken packaging [5].

Low batteries often do not result in a fail-stop behavior of a sensor node. Rather, nodes show random behavior below a certain low battery level. In the Redwood Trees deployment [12], for example, *wrong sensor readings* have been observed at low battery voltage. Even worse, a slowly degrading node can also impact the performance of other nodes in a network ensemble, possibly still having enough energy.

An increased amount of network traffic, compared to initial calculations, led to an early battery depletion due to unexpected overhearing (e.g., Great Duck Island deployment [19], section 2.1) or repeated network floods (e.g., LOFAR-agro deployment [2], section 2.8). In the Great Duck Island deployment [5], a low-resistance path between the power supply terminals was created by water permeating a capacitive humidity sensor, resulting in early battery depletion and abnormally small or large sensor readings. Poor packaging [7, 18], bad contacts [15] to sensors or deteriorating sensors [7] are typical problems encountered for sensor nodes, especially in harsh environmental conditions.

Software bugs often result in *node reboots*, for example, due to failure to restart the watchdog timer of the micro controller (e.g., LOFAR-agro deployment [2]). Also observed have been software bugs resulting in hanging or killed threads, such that only part of the sensor node software continued to operate. Overflows in counters may also deviate the sensor readings by spoiling a reference interval [18]. In [7] problems are also attributed to incorrectly downloaded software.

Sink nodes act as gateways between a sensor network and the Internet. In many applications they store and forward data collected by the sensor network to a background infrastructure. Hence, problems affecting sink nodes or the gateway must be promptly detected to limit the impact of data loss (e.g., GlacsWeb deployment [9], Great Duck Island deployment [19], Redwood Trees deployment [12]). This can also manifest in temporary errors as reported for the serial link between the sink and its secondary interface [18] or periodic problems due to watchdog reboots [11].

3.2 Link Problems

Field experiments (e.g., [6, 20]) demonstrated a very high variability of link quality both across time and space resulting in temporary link failures and variable amounts of *message loss*. Interference in office buildings can considerably affect the packet loss; the source often cannot be determined [18].

Network congestion due to *traffic bursts* is another source of message loss. In the Great Duck Island deployment [19], for example, a median message loss of 30% is reported for a single-hop network. Excessive levels of traffic bursts have been caused by accidental synchronization of transmissions by multiple senders, for example, due to inappropriate design of the MAC layer [21] or by repeated network floods as in the LOFAR-agro deployment [2]. If message loss is compensated for by retransmissions, a *high latency* may be observed until a message eventually arrives at the destination. Congestion is especially critical in dense networks, e. g. , when many nodes detect an event simultaneously and subsequently compete with the neighbors for medium access to send off an event notification [7].

Most protocols require each node in the sensor network to discover and maintain a set of network neighbors (often implemented by broadcasting HELLO messages containing the sender address). A node with *no neighbors* presents a problem as it is isolated and cannot communicate. Also, *neighbor oscillation* is problematic [21], where a node experiences frequent changes of its set of neighbors.

A common issue in wireless communication are *asymmetric links*, where communication between a pair of nodes is only possible in one direction. In a field experiment [20] between 5-15% of all links have been observed to be asymmetric, with lower transmission power and longer node distance resulting in more asymmetric links. If not properly considered, asymmetric links may result in fake neighbors (received HELLO from a neighbor but cannot send any data to it) and broken data communication (can send data to neighbor, but cannot receive acknowledgments).

Another issue is the physical length of a link. Even though if two nodes are very close together, they may not be able to establish a link (*missing short links*). On the other hand, two nodes that are very far away from each other (well beyond the nominal communication range of a node), may be able to communicate (*unexpected long links*). Experiments in [20] show that at low transmit power about 1% of all links are twice as long as the nominal communication range. These link characteristics make node placement highly non-trivial as the signal propagation characteristics of the real-world setting have to be considered [22] to obtain a well-connected network.

Most sensor network MAC protocols achieve energy efficiency by scheduling communication times and turning the radio module off in-between. Clock drift or repeated failures to re-synchronize the communication phase may result in failures to deliver data as nodes are not ready to receive when others are sending. In [23], for example, excessive *phase skew* has been observed (about two orders of magnitude larger than the drift of the oscillator).

3.3 Path Problems

Many sensor network applications rely on the ability to relay information across multiple nodes along a multi-hop path. In particular, most sensor applications include one or more sink nodes that disseminate queries or other tasking information to sensor nodes and sensor nodes deliver results back to the sink. Here, it is important that a path exists from a sink to each sensor node, and from each sensor node to a sink. Note that information may be changed as it is traversing such a path, for example due to data aggregation. Two common problems in such applications are hence *bad path to sink* and *bad path to node*. In [2], for example, *selfish nodes* have been observed that did not forward received traffic, but succeeded in sending locally generated messages.

Since a path consists of a sequence of links, the former inherits many of the possible problems from the latter such as *asymmetric paths*, *high latency*, *path oscillations*, and *high message loss*. In the Great Duck Island deployment [19], for example, a total message loss of about 58% was observed across a multi-hop network.

Finally, *routing loops* are a common problem, since frequent node and communication failures can easily lead to inconsistent paths if the software is not properly prepared to deal with these cases. Directed Diffusion [24], for example, uses a data cache to suppress previously seen data packets to prevent routing loops. If a node reboots, the data cache is deleted and loops may be created [25].

3.4 Global Problems

In addition to the above problems which can be attributed to a certain subset of nodes, there are also some problems which are global properties of a network. Several of these are failures to meet certain application-defined quality-of-service properties. These include *low data yield*, *high reporting latency*, and *short network lifetime* [26].

Low data yield means that the network delivers an insufficient amount of information (e.g., incomplete sensor time series). In the Redwood Trees deployment [12], for example, a total data yield of only about 20-30% is reported. This problem is related to message loss as discussed above, but may be caused by other problems such as a node crashing before buffered information could be forwarded [11], buffer overflows, etc. One specific reason for a low data yield is a *partitioned network*, where a set of nodes is not connected to the sink.

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

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

3.5 Discussion

Orthogonal to the classification in the previous section, the deployment problems in the surveyed literature fall into two categories: implementation and design defects. A majority of the problems reported have been caused by problems and defects in the hardware and software implementation, and can be fixed after they have been detected, analyzed, and understood. Here, dedicated inspection tools allow to find the defects quicker.

The two most underestimated problems in the surveyed sensor network deployments have been the water-proof packaging of the sensor nodes required for an outside deployment and the provision of a reliable base station which records sensor data and has to run for months and years. This suggests that sensor nodes should be sold together with appropriate packaging. However, due to the variety of sensors used for different applications, a common casing is often not practicable or possible. The provision of a reliable base station is not specific to wireless sensor networks and mostly depends on a reliable power supply and software implementation.

4 Understanding the System

To detect the problems discussed in the previous section and to identify their causes, one needs to analyze the state of the system as it changes over time. Here, the term *system* does not only refer to the state of the sensor network, but also to the state of its enclosing environment as the latter is closely coupled to the state of the sensor network. The state of the sensor network itself is comprised of the state of the individual nodes plus the states of the communication channels between the nodes. The state of a sensor node can be further refined into the hardware state and the state of the software executed by the microcontroller. Isolating the contribution of a single system component from the overall context is a complicated task and care should be taken to actually design a sensor network in a way that this task is facilitated.

As many sensor networks are real-time systems or have real-time aspects (e.g., sensor readout at regular intervals), it is not only important to understand the progression of the system state over time, but also its exact timing, i.e., *when* a certain state is assumed with respect to real time. Especially in the context of a larger distributed system, in the case of a sensor network even exacerbated by stringent resource constraints, this is a feat requiring novel tools and methods for analysis.

4.1 Hardware

The sensor node hardware typically consists of three main subsystems: the microcontroller, a low-power radio and a sensing subsystem, often implemented as an auxiliary sensor board. Components are connected through buses, e.g., UART or SPI, and interrupt lines. Each of these components executes concurrently with synchronization between contexts initiated through interrupts on the microcontroller. In the following, we discuss the individual component state of these three main building blocks and discuss how physical characteristics such as power consumption allow for inferring the overall system state.

Component State System state is typically referred to as the state of the microcontroller and is inferred using an instrumentation technique (cf. Sec. 5). However, the microcontroller is not the sole functional component attributing to the system state; the node hardware state machine is the cross-product of each of the individual component state machines. As an example consider the process of sending a packet: The microcontroller prepares the packet and sends the packet to the radio buffer, e.g., via the SPI bus. While the radio schedules the transmission based on its internal state machine, the microcontroller can either perform further work or go to a low-power sleep state. In this interval, the microcontroller has no knowledge of the ongoing state changes of the radio driver. Finally, when a packet is transmitted, the radio notifies the processor via an interrupt.

Power Consumption Each component of a sensor node can be characterized by its power consumption. Power consumption is dependent chiefly on current state but also features dynamic, non-linear effects due to capacitances and inductances. Sensor node components have typically very distinctive power consumption characteristics for their individual state. As an example, Table 3 shows power consumption for different hardware component states of the microcontroller and the radio. A detailed overview of current draw of different hardware states is provided in [27].

Microcontroller	Radio	Current (mA)
on	RX	23
on	TX	21
on	off	2.4
idle	off	1.2
standby	off	0.021

Table 3. Current consumption for different component states of a Tmote Sky node. Max values derived from datasheet.

In the context of a detailed powertrace is is possible to infer system behavior and system state in a most expressive way. Shown in Figure 1 is an exemplary powertrace of an enhanced synchronized low-power listening scheme implemented in TinyOS-2.x for the Tmote Sky platform. By timing and coordinating packet transmissions between periodic listening/acknowledgement periods (1) a data packet must not be sent repetitively (2) but only once when the receiving side is ready listening (3) [28].

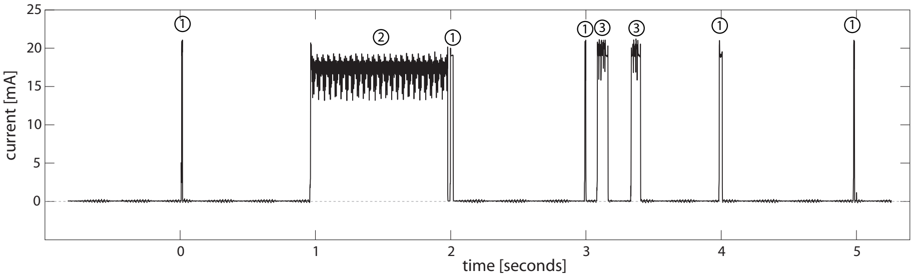

Fig. 1. A powertrace yields detailed insight into system behavior and anomalies.

4.2 Software

The state of the software executed by a microcontroller is defined by memory contents and by the contents of the registers of the microcontroller, in particular the program counter and the stack pointer. As many microcontrollers feature a Harvard architecture with separate program memory (flash) and data memory (RAM), the focus is often on the data memory as the contents of the program memory typically do not change over time. Many sensor nodes offer additional flash memories. While earlier systems used these primarily as data sinks for logging (and hence flash memory contents do not affect program behavior directly), there is a trend to use these memories in a more interactive way, for example, for file systems or virtual memories. In this case, flash memory contents may directly affect program behavior and hence are as important as RAM in understanding the behavior of the system.

Instead of working with a low-level view of the software state directly, one is often more interested in a higher-level view, e.g., the values of certain program variables (instead of raw memory contents) or a function call trace (instead of the content of the program counter). These high-level views also depend on the functionality of the operating system used. In many operating systems such as TinyOS, there is no strict separation between a kernel and the application. Here, the kernel is simply a library of functions that is linked to the application code. As there is no means of isolation and protection, bugs in the application code can modify kernel state and vice versa, such that the complete software state has to be taken into account to understand the behavior of the system.

Other systems offer a more strict separation and sometimes also isolation mechanisms between kernel and application. If this is the case, then one may trust that the operating system is well-behaved and focus on the application state. Some operating systems provide a thread abstraction with an individual program counter per thread. Although threads offer an additional level of abstraction, they also introduce additional complexity due to interactions between threads (e.g., shared variables, synchronization, timing aspects of thread scheduling).

4.3 Communication

The contents and timing of messages exchanged between sensor nodes does not only disclose the nature of interactions between sensor nodes (e.g., neighborhood relationships, routing topologies), but does also provide hints on the state of individual nodes. For example, a lack of messages from a certain node may indicate that the node has died, or the values of a sequence number contained in a message may indicate certain problems such as node reboots (after which the sequence number counter in the node memory is reset to an initial value). Often dedicated messages carry system health information that is vital to the deployment, commissioning and operation of a sensor network application. As a number of deployment reports have shown, e.g., [29], expressiveness and simple, cleartext access to message payload is preferential over elaborate coding or even segmentation of data. In the case of segmentation of context over multiple packets, state cannot be reconstructed if one of the packets gets lost.

4.4 Environment

Sensor networks are deeply embedded into the physical world and hence the behavior of a sensor network is strongly dependent on the state of the environment. Not only does the state of the environment control the output of sensors attached to sensor nodes, but it also influences the wireless communication channel and hence communication as well as hardware performance (e.g., frequency of oscillators). Hence, additional infrastructure can be used (e.g., video cameras as in [5, 30]) to measure the state of the environment in a given deployment which is not only important to obtain ground truth for calibration of the system and individual sensors, but also to understand the reasons of certain failures, system behavior in general or degraded performance.

Some sensor networks apply techniques to process or aggregate sensor data in the network, such that the data collected from the sensor network does not disclose the output of individual sensors. In such settings it may be helpful to provide access to the raw output of the sensors in order to understand the system behavior.

5 Node Instrumentation

Node instrumentation is concerned with modifications of sensor nodes (both at the hardware and software levels) to allow access to the system state. One fundamental challenge here is to minimize interference with the application in the sense that introducing instrumentation should avoid so-called *probe effects*, where instrumentation (i.e., inserting a probe) results in a changed behavior of the system.

5.1 Software Instrumentation

Software instrumentation is required to retrieve the state of the software executing on the microcontroller. In practice, it is impossible (since there is not enough bandwidth to extract this information from the sensor node) and often also not necessary to extract the complete software state continuously. Instead, only a slice of the state (e.g., the values of a certain set of program variables) at certain points in time (e.g., when a certain program variable is modified, when a certain function is entered) suffices in many cases.

Source vs. Binary Software instrumentation can be achieved at two levels: at the source level, where the source code of the software is modified, or at the binary level, where the binary program image (i.e., output of the assembler) is modified. At the source level, calls to specific functions are inserted, for example to log the value of a certain variable when a new value is assigned to this variable [31]. At the binary level, the binary CPU instructions are modified. A particular difficulty with this approach is that it is not easily possible to insert additional instructions as the code contains references (e.g., jump instructions) to certain addresses in the code image. By inserting instructions, these references would be invalidated. A common technique to avoid these problems are so-called *trampolines* [32–34], where an instruction X at address A in the code image is overwritten with a jump instruction to an address after the end of the binary image. At that address, a copy of the overwritten instruction X is placed, followed by instructions to retrieve the software state, followed by a jump instruction back to address A+1. This way, the layout of the original binary code image is not changed as the code to retrieve the software state is appended to the code image.

In some cases, instrumentation of the binary can be performed without actually modifying the binary code, by linking in additional code that makes references to the symbols (e.g., of variables or functions) defined in the binary. Marionette [35], for example, uses this approach to link RPC stubs with the binary to allow remote access to variables and functions in the code executing on the sensor node. The stubs are created by parsing an annotated version of the source code of the application.

Instrumentation at the source level is often easier to implement, but may significantly change the resulting binary code. For example, due to the inserted function calls the compiler may decide for a different optimization strategy, resulting in a significantly different binary code image being created. This may lead to pronounced *probe effects*, where the instrumented software behaves very different from the unmodified software due to changed memory layout, register allocations, or timing issues. In contrast, binary instrumentation is more complex to implement because one has to operate at the level of machine instructions, but probe effects are often less significant because the basic memory layout is not changed.

Operating System vs. Application Another aspect of software instrumentation is whether the actual application code and/or the operating system is instrumented. In some cases it may be sufficient to only instrument the operating system or runtime environment to trace timing and parameters of kernel invocations, memory allocations, or context switches. In particular when using a virtual machine to interpret a byte code representation of the application program, an instrumentation of the byte code interpreter provides a detailed view of the applications state without actually instrumenting the application code. Some virtual machines have been developed for sensor nodes (e.g., [36]), but constrained resources often preclude the use of virtual machines low-end sensor nodes due to the resulting performance degradation.

Dynamic Instrumentation Sometimes it is necessary to change the instrumentation during runtime, for example to implement interactive debugging, where one wants to place watchpoints that are triggered when the value of a certain variable changes. While this is typically impossible with source code instrumentation as code would have to be recompiled and uploaded to the microcontroller, virtual machines and binary instrumentation do support this. A difficulty with binary instrumentation is that the program image resides in flash memory rather than in RAM. Changing the contents of flash memory is not only slow and consumes a lot of energy, but flash only supports a limited number of write cycles (in the order of thousands) before it fails [34]. Hence, changes of the binary instrumentation during runtime should be rather infrequent. This problem does not occur with virtual machines, as the interpreted program typically resides in data memory (i.e., RAM) – only the virtual machine itself resides in flash memory.

Specifications One further aspect of software instrumentation is the way the user specifies how the software should be instrumented, i.e., which slice of the software state should be retrieved at what point in time. The most basic way is to specify the exact locations in the program code (either at source or binary level) where an instrumentation should be placed and which slice of the program state this instrumentation should retrieve. However, this is often a tedious and error-prone process. For example, when monitoring the value of a certain program variable over time, one may forget to place an instrumentation after one of the many places in the program where the value of this variable may be changed.

A more advanced approach inspired by Aspect-Oriented Programming (AOP) is to let the user specify a certain pattern such that instrumentation is inserted wherever the program code matches this pattern [32]. Example patterns may be of the form “whenever a function named *set** is entered, insert a given instrumentation” or “whenever the value of a variable named *state** is modified, insert a given instrumentation”. An interpreter would read these patterns, match them with the program code (either source or binary) and place the appropriate instrumentation.

A related approach is based on the notion of events [37]. Here, an event is said to occur when the software assumes a certain predefined state. Instrumentation has to be added at all points where the software enters a state that matches the event definition. One basic way to implement this is to let the user identify all these points in the program. But in principle, this can also be automated. Upon occurrence of an event, a notification is generated that identifies the event, often also containing the time of occurrence and a certain slice of the software state at the time of occurrence of the event. One example of such events are watchpoints in a debugger [34].

Systems that support pattern-based or event-based instrumentation during runtime typically include a small runtime system to evaluate the pattern or event specification in order to identify the points in the code where instrumentation has to be placed, and to perform the actual instrumentation [32–34].

5.2 Hardware Instrumentation

Hardware instrumentation enables the extraction of the hardware state but also of the software state from a sensor node. Note that a hardware instrumentation is not strictly necessary. For example, one may send off the software state (extracted using the techniques described in the previous section) using the radio or by blinking LEDs. Here, one can place an additional radio receiver or a video camera next to the sensor nodes to capture the state without physical access to the sensor nodes. In some cases it is even possible to retrieve parts of the hardware state in this way. For example, one may use the built-in analog-to-digital converter of the microcontroller to measure the battery voltage and send it off via radio or LEDs. However, all these techniques may result in significant changes of the system behavior and thus may cause probe effects. Hardware instrumentation tries to minimize these effects, but requires physical access to the sensors nodes, which may be difficult in the field.

Hardware instrumentation of sensor nodes uses the approaches and mechanisms known from embedded systems [38]. However, for WSN there is a considerable distinction: a single embedded system is only part of an interacting network. Thus in the following we describe only approaches from debugging embedded systems, which are focussed on a system aspect. Traditional embedded debugging methods for a single node such as flash monitors are not considered. We focus on hardware instrumentation utilizing a physical connection to the sensor node to extract logical monitoring information. On the sensor nodes, microcontroller interfaces are used to drive signals across a wired connection to a monitoring observer. The connection to an observer may include a converter translating the serial protocol to a standard interface such as USB or Ethernet.

The simplest approach for hardware instrumentation is to use General-Purpose IO's to generate binary flags, representing internal state of the node. While information content is limited, such wired pins are very helpful in debugging and have a minimal overhead in software. Further information can be extracted by using a serial protocol over the wire. Many microcontrollers provide support for serial protocols in their interfaces; in particular UARTs (universal asynchronous receiver/transmitters) are typically included. These interfaces can be used to send monitoring messages from the node to an observer. The additional information content is payed by increased, non-deterministic latency of an information message due to buffering and serialization. Flow control and error handling may be incorporated depending on monitoring message frequency and instrumentation requirements.

Additionally, microcontrollers typically include JTAG (IEEE 1149.1) functionality. JTAG stops the microcontroller and subsequently allows for non-intrusive extraction of node state to the external observer. [1] While, this method is very helpful in debugging

[1] Note that our discussion only focusses on information extraction and not on the debugging capabilities of JTAG such as stepping through a program execution.

individual nodes, the stopping of the execution induces considerable probing effects due to the halting of interaction with the system.

A separate HW instrumentation technique for extracting physical information is observation of power consumption through voltage and current measurements. While this is traditionally performed with expensive lab equipment such as voltage/current meters or oscilloscopes, in [39] a low-cost and distributed hardware instrumentation for power measurements on a testbed is presented. As sensor nodes have significant differences in power consumption for the microcontroller and the radio, the power consumption trace allows for extracting information on system state. A different approach is to focus on energy consumption of different operations. Using specialized hardware on each sensor node [40], energy attribution to individual tasks [27] may be derived allowing for additional insights.

6 Network Instrumentation Methods

A number of methods have been developed in recent years to aid development, testing and deployments of whole sensor network systems. The sheer number of nodes requires a careful and orchestrated instrumentation in order to allow to interact with a whole network of nodes. In this sense, *network instrumentation* refers to the instrumentation of sensor nodes to extract a comprehensive view of a sensor network. There are four different mechanisms for network instrumentation (i) in-band collection, (ii) local logging, (iii) sniffing and (iv) out-of-band collection. These approaches will be discussed in the following.

Approach	Online	Data Volume	Attachment
In-Band	Yes	Low	No
Storage	No	Low	No
On-line Sniffing	Yes	Low	No
Off-line Sniffing	No	Low	No
Out-of-Band	Yes	High	Yes

Table 4. Characteristics of network instrumentation methods based on distinct parameters: online capabilities, supported data volume and physical attachment.

With *in-band collection* as shown in Fig. 2 monitoring messages are routed through the sensor network to a sink, where they are merged and fed to the evaluation backend. This is similar to a data gathering approach to collect sensor values from the network. 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.

With the *logging* approach in Fig. 2 (b), monitoring messages are stored to a local node memory (flash) 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. As

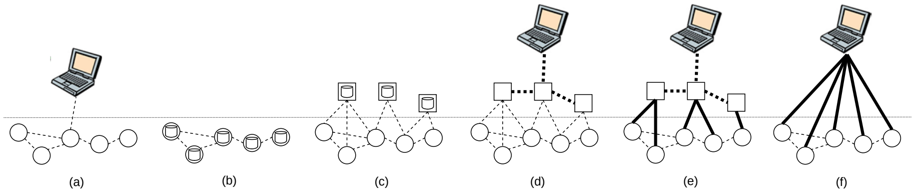

Fig. 2. Different network instrumentation methods have implications on resources used and the expressiveness of the results: (a) in-band collection (b) logging (c) offline sniffing (d) online sniffing (e) wireless testbed (f) wired testbed.

local storage is typically limited by sensor nodes, the amount of monitoring messages is considerably limited.

With the *offline sniffer* approach in Fig. 2 (c), an additional set of sniffer nodes (depicted as squares) is installed alongside the sensor network to overhear messages messages sent by sensor nodes. Sniffer nodes store these messages in their (flash) memories. As in the logging approach, sniffer nodes need to be collected to download messages from their memories, while the sensor network remains operational. This approach may be used in a completely passive manner without modifying sensor network software or protocols as in [41]. However, one may also modify the sensor network protocols to include additional information about node states into messages or to broadcast additional messages in-band with the application traffic to be overheard by the sniffer nodes. However, other sensor nodes would ignore these additional messages and would not forward them, resulting in significantly less in-band traffic compared to the in-band collection.

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 monitoring messages to a dedicated 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 [42].

Out-of-band logging avoides completely avoids the use of the sensor node radio for monitoring purposes to as to minimize interference with the application. Instead, a second, “out-of-band” communication channel is created by wiring the sensor nodes. However the implementation may differ concerning the transport mechanism for monitoring messages. A *wireless testbed* approach in Fig. 2 (e) is similar to the online sniffer, but instead of sending monitoring messages in-band with application traffic, each sensor node is connected by wire (e.g., serial cable) to an additional node, a so-called observer 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 [43]. Finally, with the *wired testbed* approach in Fig. 2 (f), each node is wired to a sink. Monitoring messages 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, e.g., [44]. In fact, wired testbeds are the most common instrumentation method used to date. However, this approach is typically only feasible in the lab, while the other approaches could be applied both in the lab and in the field.

A basic concept in all instrumentation methods is that of an *observer*, be it integrated as in the in-band case, or external as in the out-of-band case. Any state that wants to be observed needs to be preserved and made available for extraction and further analysis. By doing so, any method used ultimately influences the system it is applied on. Even in the case of a passive sniffer, care must be taken that the relevant data necessary for successful observation and analysis is actually exported. Here, a protocol designer thus must foresee provisions in the protocol design that allow to reconstruct the necessary context. Additionally, all network instrumentation methods rely on successful node instrumentation, especially w.r.t the preservation of timing properties. As stated earlier, wired testbeds are the most commonly used approach today. However, most testbeds are limited to distinct areas only, e.g., an office building. As testing has to take into account the influence of the actual environment to be encountered at the deployment site, special equipment is required to simulate environmental impact, for example by incorporating a temperature cycling chamber into a testbed setup.

7 Analyzing the System

Once access to the system state is provided through node and network instrumentation, system state can be analyzed, e.g. to detect failures. This analysis can be performed within the network on the sensor nodes themselves, outside of the sensor network, or with a mixture of both approaches (e.g., some preprocessing is performed in the sensor network to reduce the amount of information that has to be transmitted).

A fundamental problem that has to be addressed in analyzing the system state is incomplete information. Not only do limited node and network resources preclude to extract the complete system state, but often wireless communication with inherent message loss is used to extract the system state.

7.1 Monitoring and Visualization

Visualization of system parameters and state is an invaluable tool for immediate feedback. A number of traditional network analysis tools such as Ganglia, Cacti or Nagios can be readily applied [45], but especially at the deployment site handheld visual inspection devices [16, 18] provide insight into the operation of the network and individual nodes. As an example, SeeDTV [16] provides a handheld device, SeeMote, providing visual support, while being small, light-weight, and providing long battery lifetime. The SeeDTV application provides different views on the information provided by the sensor network such as statistics on the number of available nodes, status and health (battery level) of an individual selected node, and a detailed view onto the node's sampled ADC values. SeeDTV was used in the Luster deployment and helped to pinpoint 5 malfunctioning nodes, which could be immediately replaced.

With a focus on the development stage, Octopus [46] is an interactive tool allowing for visualizing information of system characteristics of a sensor network. It additionally provides an interface for interactive reconfiguration of application parameters. As an example, a user may change the duty-cycle and observe the effects on the network.

7.2 Inferring Network State from Node States

Often a user is overwhelmed by monitoring the low-level system state extracted from the sensor nodes. Here, the low-level traces from multiple nodes can be combined to interfere a higher-level state of the network. For example, instead of looking at the individual routing tables of nodes, one may reconstruct the routing topology of the network and display it to the user.

SNIF [42] allows to infer the network state from message traces collected with a sniffer network. Inference is implemented with a data stream framework. The basic element of this framework is a data stream operator which accepts a data stream (e.g., a stream of overheard messages) as input, processes the stream (e.g., by removing elements from the stream or modifying their contents), and outputs another data stream. These operators can be chained together to form a directed acyclic graph. There are general-purpose operators that can be configured with parameters (e.g., a union operator that merges two data streams) and custom operators which are implemented by a user for a specific inference task. Ideally, one should be able to implement a given inference task just by configuring and combining existing general-purpose operators. However, in practice it is often necessary to implement some custom operators. LiveNet [41] provides a similar functionality, however the inference functionality is implemented in an ad-hoc manner using a general-purpose programming language.

EvAnT and the successive Rupeas DSL [37, 47] provide an event analysis framework for WSNs. Input to the system is a trace of log events collected from the execution. Each event is a tuple of key-value pairs, minimally including a node identifier and a type. Traces are represented as sets of events with no particular ordering on events implied. Rather the user may specify a strict ordering using timestamps or a partial order based on message ordering on sent and received packets. EvAnT allows for formulating queries and assertions on the trace. Queries are based on the EvAnT operators, which use simple predicates and associating functions for combining events. As an example, a routing path is composed by iteratively associating individual send/receive and forward links across the nodes. A case study in [37] presents how concise formulations in EvAnT extract information about message flow. In particular, through subsequent queries on the trace, the underlying problem of a low-power data gathering application is traced back to the time synchronization protocol.

7.3 Failure Detection

Failure detection is concerned with automatically identifying system states that represent failures (i.e., deviations from the system specification). Hence, instead of manually scanning traces of node or network states, problematic system states are automatically identified.

There are two orthogonal approaches to failure detection. Firstly, one can specify the correct behavior of the system and deviations from this correct behavior are automatically detected and assumed to be failures. Secondly, one can directly specify faulty behavior and the actual system state is then matched with the failure specifications.

The first approach has the advantage that potentially any failure can be detected, even ones that are unknown a priori. However, it is often non-trivial to specify the correct behavior of the system as a wide range of different systems states may represent correct behavior. This is especially true for failures that result from interactions of multiple nodes, because here one has to specify the correct behavior of the whole network. In contrast, specifying the correct behavior of a single node is typically much easier.

One approach to deal with this problem is to focus on certain aspects of the system behavior instead of trying to specify the complete system behavior. One example for this approach are assertions, where the user can specify a predicate over a slice of the system state, formulating the hypothesis that this predicate should always be true. If the predicate is violated, a failure is detected. Note, however, that the user may also specify an incorrect hypothesis, so a failed assertion does not necessarily indicate a failure. Assertions can be non-local both in time (refer to system state obtained at different points in time) and space (refer to the state of multiple nodes). For example, passive distributed assertions [31] are local in time but non-local in space. Other examples for the use of assertions are [31, 33, 37, 48].

Another approach to address the problem of specifying the correct behavior of a system is to use machine learning techniques to automatically build a model of the system state during periods when the system is known to behave correctly. In Dustminer [49], for example, the frequency of occurrence of certain sequences of events is computed while the system is known to work correctly. If at a later point in time the frequency of a certain event sequence deviates significantly from the "correct" frequency, a failure is detected.

It is often easier to directly specify the system behavior that represents a very specific failure. However, with this approach only failures can be detected that are known a priori. Many systems build on this approach. For example, SNIF [42] and Sympathy [50] both offer failure detectors for node crash, node reboot, isolated nodes with no neighbors, no path to/from sink, and others.

7.4 Root Cause Analysis

It is often the case that a failure may cause a secondary failure. For example, if a node crashes (primary failure), then this may lead to a situation where other nodes fail to route data to the sink (secondary failure) because the crashed node was the only connection to the sink. Here, a user would be interested in knowing the primary failure (i.e., the root cause) instead of being overwhelmed with all sorts of secondary failures.

One technique that has been applied to sensor networks for root cause analysis are so-called decision trees [50, 42]. As illustrated in Fig. 3 taken from [50], a decision tree is a binary tree where each internal node is a detector for a specific failure that may either output *Yes* (failure detected) or *No* (failure not detected). The leaf nodes indicate possible failures that can be detected. To process the decision tree, we start at the root and execute the failure detector. Depending on the output of the failure detector, we proceed to one of the child nodes. If the child node is a leaf, the failure noted in the leaf node is output and we are done. Otherwise, if the node is an inner node, we evaluate the respective failure detector and so on until we arrive at a leaf node.

Decisions trees in [50, 42] are empirically constructed, according to the rule that if failure A can result in a secondary failure B, then the failure detector node for A should be located on the path from the root of the tree to the node representing the failure detector for B.

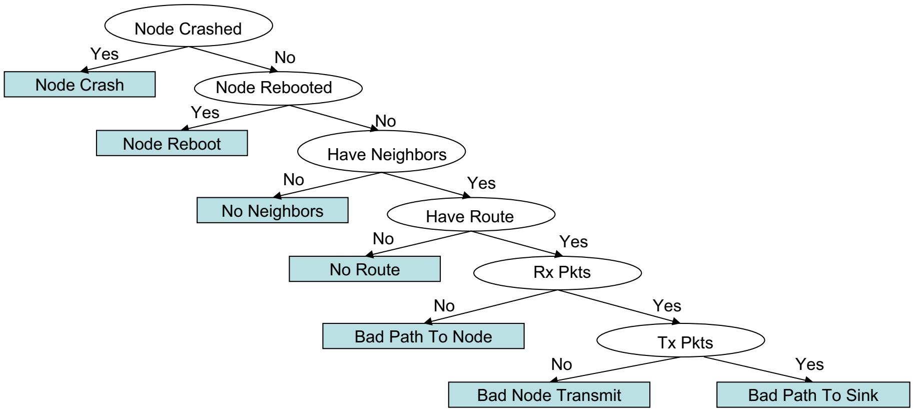

Fig. 3. The decision tree for root cause analysis used by Sympathy [50].

7.5 Node-Level Debugging

Traditional source-level debugging techniques such as placing breakpoints, watchpoints, or single-stepping can also be applied to sensor nodes. For example, in [34], the GNU debugger gdb is ported to TinyOS. However, this approach is limited as a sensor node is merely part of a larger network of nodes that often operate under real-time constraints. Since the timing of program execution is significantly affected by the debugger, significant probe effects are caused. In the extreme case, the slowed-down execution of the debugged node may itself lead to failures. Another problem with this approach is that dynamic modifications of the binary code executing on the microcontroller are needed to implement the debugger. As the program image is stored in flash memory, this leads to excessive wear of the flash memory and high latency.

7.6 Replay and Checkpointing

Another analysis technique is to reproduce a faulty execution either directly in the deployed network or in a controlled environment (e.g., lab testbed or simulator). By varying some of the system parameters during such a replay, one can examine potential dependencies between these parameters and the faulty behavior, verify that a certain modification of the system has removed the cause of a problem, or simply tune the performance of the system.

Envirolog [51] supports repeatability of system executions albeit asynchronous events. This is grounded on the assumption of scoped event readings, i.e., data is only transferred via dedicated functions and associated parameters. A log module is responsible for logging events of interest, i.e., the according function calls with their parameters into flash memory, thus creating a timestamped event record. In a subsequent replayed execution, consumers of these events are re-wired to connect to the replay module, providing the recorded events from flash at the replayed instant in time. While this allows for repeated tests of different parameter settings, e.g., for protocol tuning, the results are only valid as long as the re-executions do not change the occurrence and timing of events. This fundamental problem of timing dependence is avoided with a model-based approach such as Emuli. Emuli [52] focusses on the capability to emulate network-wide sensing data to sensor nodes. This allows for analyzing the system execution on a model-based ground truth, whether it is used for repeatability or exploratory analysis for novel sensing data. Emuli provides time synchronization and coordination among nodes in order to generate a consistent network wide model of sensing data.

Checkpointing [53] extracts system state allowing for analysis, transferral between different execution (test) platforms, e.g., a simulator and a testbed, and for repeatability. However, in contrast to replay repeatability discussed above, checkpointing only concerns the initial state of an execution, not the subsequent execution itself. Nevertheless, a consistent state across the system, e.g., concerning the topology and neighbor tables in network-centric tests, is invaluable for comparative tests and analysis of parameter changes. Checkpointing of a single sensor node concerns the state of the individual components. As program code in program flash is typically not modified, only RAM state needs to be extracted, which is typically already provided by the bootloader. Other components such as External Flash, LEDs, and especially the radio require special considerations dependent on the requirements of application and the checkpoint. Dunkels et al. [53] describe the support in Contiki for storing and loading node state and its current scope. However, the difficulty of checkpointing lies in the synchronized checkpointing across all nodes in the network for a consistent snapshot. This is handled by intermittent Linux routers controlling a subset of nodes for freezing and unfreezing.

8 Concluding Remarks

In this contribution we have surveyed the most prominent sensor network deployments and been able to identify selected underlying problems in system design, during installation and deployments. A special focus has been made towards problems arising and relevant in practice with sensor network applications and deployments in a real environment. We have presented a number of techniques for the instrumentation and analysis, predominantly to be applied at run-time of a sensor network. With the techniques presented here, a future designer of networked embedded systems should be able to define a suitable and successful design strategy suitable to meet requirements specified. With regard to the distributed nature of sensor networks the typical problems encountered have been classified into node, link, network path as well as global problems. Likewise the methods presented in this article are targeting to further understanding at the node, network and system level. Due to the diversity of applications, requirements and design goals, there is no single, distinctive approach to the design and deployment of sensor networks available today.

References

1. Polastre, J.P., Szewczyk, R., Mainwaring, A., Culler, D., Anderson, J.: Analysis of wireless sensor networks for habitat monitoring. In Raghavendra, C.S., Sivalingam, K.M., Znati, T., eds.: Wireless Sensor Networks. Kluwer Academic Publishers (2004)

2. Langendoen, K., Baggio, A., Visser, O.: Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In: Proc. 20th Int'l Parallel and Distributed Processing Symposium (IPDPS 2006), IEEE, Piscataway, NJ (April 2006) 8–15 14th Int'l Workshop Parallel and Distributed Real-Time Systems (WPDRTS 2006).
3. Polastre, J., Szewczyk, R., Culler, D.: Telos: Enabling ultra-low power wireless research. In: Proc. 4th Int'l Conf. Information Processing Sensor Networks (IPSN '05), IEEE, Piscataway, NJ (April 2005) 364–369
4. Dubois-Ferrière, H., Fabre, L., Meier, R., Metrailler, P.: Tinynode: a comprehensive platform for wireless sensor network applications. In: Proc. 5th Int'l Conf. Information Processing Sensor Networks (IPSN '06), ACM Press, New York (2006) 358–365
5. Szewczyk, R., Polastre, J., Mainwaring, A., Culler, D.: Lessons from a sensor network expedition. In: Proc. 1st European Workshop on Sensor Networks (EWSN 2004). Volume 2920 of Lecture Notes in Computer Science., Springer, Berlin (January 2004) 307–322
6. Woo, A., Tong, T., Culler, D.: Taming the underlying challenges of reliable multihop routing in sensor networks. In: Proc. 1st ACM Conf. Embedded Networked Sensor Systems (SenSys 2003), ACM Press, New York (2003) 14–27
7. Arora, A., Dutta, P., Bapat, S., Kulathumani, V., Zhang, H., Naik, V., Mittal, V., Cao, H., Demirbas, M., Gouda, M., Choi, Y., Herman, T., Kulkarni, S., Arumugam, U., Nesterenko, M., Vora, A., Miyashita, M.: A line in the sand: a wireless sensor network for target detection, classification, and tracking. Computer Networks **46**(5) (2004) 605 – 634
8. Tateson, J., Roadknight, C., Gonzalez, A., Fitz, S., Boyd, N., Vincent, C., Marshall, I.: Real world issues in deploying a wireless sensor network for oceanography. In: Proc. Workshop on Real-World Wireless Sensor Networks (REALWSN 2005). (June 2005)
9. Padhy, P., Martinez, K., Riddoch, A., Ong, H.L.R., Hart, J.K.: Glacial environment monitoring using sensor networks. In: Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN). (2005)
10. Paek, J., Chintalapudi, K., Govindan, R., Caffrey, J., Masri, S.: A wireless sensor network for structural health monitoring: Performance and experience. In: Proc. 2nd IEEE Workshop on Embedded Networked Sensors (EmNets 2005). (May 2005) 1–9
11. Stoianov, I., Nachman, L., Madden, S., Tokmouline, T.: Pipenet - a wireless sensor network for pipeline monitoring. In: Proc. 6th Int'l Conf. Information Processing Sensor Networks (IPSN '07), ACM Press, New York (2007) 264–273
12. Tolle, G., Polastre, J., Szewczyk, R., Culler, D., Turner, N., Tu, K., Burgess, S., Dawson, T., Buonadonna, P., Gay, D., Hong, W.: A macroscope in the redwoods. In: Proc. 3rd ACM Conf. Embedded Networked Sensor Systems (SenSys 2005). (November 2005) 51–63
13. Madden, S., Franklin, M., Hellerstein, J., Hong, W.: TinyDB: An acqusitional query processing system for sensor networks. ACM Transactions on Database Systems **30**(1) (2005) 122–173
14. Werner-Allen, G., Lorincz, K., Johnson, J., Lees, J., Welsh, M.: Fidelity and yield in a volcano monitoring sensor network. In: Proc. 7th Symp. Operating Systems Design and Implementation (OSDI '06), ACM Press, New York (2006) 381 – 396
15. Selavo, L., Wood, A., Cao, Q., Sookoor, T., Liu, H., Srinivasan, A., Wu, Y., Kang, W., Stankovic, J., Young, D., Porter, J.: Luster: wireless sensor network for environmental research. In: Proc. 5th ACM Conf. Embedded Networked Sensor Systems (SenSys 2007), ACM Press, New York (2007) 103–116
16. Liu, H., Selavo, L., Stankovic, J.: SeeDTV: deployment-time validation for wireless sensor networks. In: Proc. 4th IEEE Workshop on Embedded Networked Sensors (EmNetS-IV), ACM Press, New York (2007) 23–27
17. Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M., Couach, O., Parlange, M.: Sensorscope: Out-of-the-box environmental monitoring. In: Proc. 7th Int'l Conf. Information Processing Sensor Networks (IPSN '08), IEEE CS Press, Los Alamitos, CA (2008) 332–343

18. Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M.: The hitchhiker's guide to successful wireless sensor network deployments. In: Proc. 6th ACM Conf. Embedded Networked Sensor Systems (SenSys 2008), ACM Press, New York (2008) 43–56
19. Szewcyk, R., Mainwaring, A., Polastre, J., Anderson, J., Culler, D.: An analysis of a large scale habitat monitoring application. In: Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004), Baltimore, Maryland, USA, ACM Press, New York (November 2004) 214 – 226
20. Ganesan, D., Krishnamachari, B., Woo, A., Culler, D., Estrin, D., Wicker, S.: Complex behavior at scale: An experimental study of low-power wireless sensor networks. Technical Report CSD-TR 02-0013, UCLA, Los Angeles, California, USA (2002)
21. Ramanathan, N., Kohler, E., Estrin, D.: Towards a debugging systems for sensor networks. International Journal of Network Management **15**(4) (July 2005) 223–234
22. Burns, R., Terzis, A., Franklin, M.: Design tools for sensor-based science. In: Proc. 3rd IEEE Workshop on Embedded Networked Sensors (EmNetS-III), Cambridge, Massachusetts, USA (May 2006)
23. Mainwaring, A., Culler, D., Polastre, J., Szewczyk, R., Anderson, J.: Wireless sensor networks for habitat monitoring. In: Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications (WSNA), Atlanta, Georgia, USA (September 2002) 88 – 97
24. Intanagonwiwat, C., Govindan, R., Estrin, D., Heidemann, J., Silva, F.: Directed diffusion for wireless sensor networking. IEEE/ACM Trans. Netw. **11**(1) (2003) 2–16
25. Sobeih, A., Viswanathan, M., Marinov, D., Hou, J.C.: Finding bugs in network protocols using simulation code and protocol-specific heuristics. In: Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM), Manchester, United Kingdom, Springer (2005) 235–250
26. Tilak, S., Abu-Ghazaleh, N.B., Heinzelman, W.R.: A taxonomy of wireless micro-sensor network models. ACM SIGMOBILE Mobile Computing and Communications Review (MC2R) **6**(2) (April 2002) 28–36
27. Fonseca, R., Dutta, P., Levis, P., Stoica, I.: Quanto: Tracking energy in networked embedded systems. In: Proc. 9th Symp. Operating Systems Design and Implementation (OSDI '08), ACM Press, New York (2008) 323–338
28. Lim, R., Woehrle, M., Meier, A., Beutel, J.: Harvester energy savings through synchronized low-power listening. In: Proc. EWSN 2009 - Demos/Posters Session, Department of Computer Science, University College Cork, Ireland (February 2009) 29–30
29. Talzi, I., Hasler, A., Gruber, S., Tschudin, C.: PermaSense: investigating permafrost with a WSN in the Swiss Alps. In: Proc. 4th IEEE Workshop on Embedded Networked Sensors (EmNetS-IV), ACM Press, New York (2007) 8–12
30. Bonnet, P., Leopold, M., Madsen, K.: Hogthrob: towards a sensor network infrastructure for sow monitoring (wireless sensor network special day). In: Proc. Conf. Design, Automation and Test in Europe (DATE 2006), IEEE, Piscataway, NJ (March 2006)
31. Römer, K.: PDA: Passive distributed assertions for sensor networks. In: Proc. 8th Int'l Conf. Information Processing Sensor Networks (IPSN '09), ACM Press, New York (April 2009)
32. Cao, Q., Abdelzaher, T.F., Stankovic, J.A., Whitehouse, K., Luo, L.: Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In: Proc. 6th ACM Conf. Embedded Networked Sensor Systems (SenSys 2008), ACM Press, New York (November 2008) 85–98
33. Tavakoli, A., Culler, D., Shenker, S.: The case for predicate-oriented debugging of sensornets. In: Proc. 5th ACM Workshop on Embedded Networked Sensors (HotEmNets 2008). (June 2008)

34. Yang, J., Soffa, M.L., Selavo, L., Whitehouse, K.: Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: Proc. 5th ACM Conf. Embedded Networked Sensor Systems (SenSys 2007), ACM Press, New York (November 2007) 189–203
35. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., Culler, D.: Marionette: using RPC for interactive development and debugging of wireless embedded networks. In: Proc. 5th Int'l Conf. Information Processing Sensor Networks (IPSN '06), ACM Press, New York (April 2006) 416–23
36. Levis, P., Culler, D.E.: Maté: a tiny virtual machine for sensor networks. In: Proc. 10th Int'l Conf. Architectural Support Programming Languages and Operating Systems (ASPLOS-X), ACM Press, New York (2002) 85–95
37. Woehrle, M., Plessl, C., Lim, R., Beutel, J., Thiele, L.: EvAnT: Analysis and checking of event traces for wireless sensor networks. In: Proc. Int'l Conf. Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2008). (June 2008) 201–208
38. Ganssle, J.G.: The Art of Programming Embedded Systems. Academic Press, Inc., Orlando, FL, USA (1992)
39. Haratcherev, I., Halkes, G., Parker, T., Visser, O., Langendoen, K.: PowerBench: A Scalable Testbed Infrastructure for Benchmarking Power Consumption. In: Int. Workshop on Sensor Network Engineering (IWSNE). (2008)
40. Dutta, P., Feldmeier, M., Paradiso, J., Culler, D.: Energy metering for free: Augmenting switching regulators for real-time monitoring. In: Proc. 7th Int'l Conf. Information Processing Sensor Networks (IPSN '08), ACM Press, New York (April 2008) 283–294
41. Chen, B., Peterson, G., Mainland, G., Welsh, M.: Livenet: Using passive monitoring to reconstruct sensor network dynamics. In: Proceedings of the 4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2008), Santorini Island, Greece (June 2008) 79–98
42. Ringwald, M., Römer, K., Vitaletti, A.: Passive inspection of wireless sensor networks. In: Proceedings of the 3rd International Conference on Distributed Computing in Sensor Systems (DCOSS 2007). (2007)
43. Dyer, M., Beutel, J., Kalt, T., Oehen, P., Thiele, L., Martin, K., Blum, P.: Deployment Support Network - A toolkit for the development of WSNs. In: Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN), Delft, The Netherlands (2007) 195–211
44. Werner-Allen, G., Swieskowski, P., Welsh, M.: Motelab: a wireless sensor network testbed. In: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN), Los Angeles, California, USA (2005) 483–488
45. Beutel, J., Dyer, M., Martin, K.: Sensor network maintenance toolkit. In: Proc. 3rd European Workshop on Sensor Networks (EWSN 2006). (February 2006) 58–59
46. Ruzzelli, A., Jurdak, R., Dragone, M., Barbirato, A., O'Hare, G., Boivineau, S., Roy, V.: Octopus: A dashboard for sensor networks visual control. In: MobiCom 2008. (2008)
47. Woehrle, M., Plessl, C., Thiele, L.: Poster abstract: Rupeas - an event analysis language for wireless sensor network traces. In: Adjunct Proc. 6th European Workshop on Sensor Networks (EWSN 2009), Cork, Ireland (February 2009) 19–20
48. Lodder, M., Halkes, G.P., Langendoen, K.G.: A global-state perspective on sensor network debugging. In: Proc. 5th ACM Workshop on Embedded Networked Sensors (HotEmNets 2008). (June 2008)
49. Khan, M.M.H., Le, H.K., Ahmadi, H., Abdelzaher, T.F., Han, J.: Dustminer: troubleshooting interactive complexity bugs in sensor networks, ACM Press, New York (November 2008) 99–112
50. Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., Estrin, D.: Sympathy for the sensor network debugger. In: Proceedings of the 3rd ACM Conference on Embedded Networked Sensor Systems (SenSys), Los Angeles, California, USA (November 2005) 255 – 267

51. Luo, L., He, T., Zhou, G., Gu, L., Abdelzaher, T.F., Stankovic, J.A.: Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In: INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings. (April 2006) 1–14
52. Alam, N., Clouser, T., Thomas, R., Nesterenko, M.: Emuli: model driven sensor stimuli for experimentation. In: Proc. 6th ACM Conf. Embedded Networked Sensor Systems (SenSys 2008), ACM Press, New York (2008) 423–424
53. Oesterlind, F., Dunkels, A., Voigt, T., Tsiftes, N., Eriksson, J., Finne, N.: Sensornet checkpointing: Enabling repeatability in testbeds and realism in simulators. In: Proc. 6th European Conference on Wireless Sensor Networks (EWSN 2009). Volume 5432 of Lecture Notes in Computer Science., Springer, Berlin (February 2009) 343–357