LiveNet: Using Passive Monitoring to Reconstruct Sensor Network Dynamics

Bor-rong Chen, Geoffrey Peterson, Geoff Mainland and Matt Welsh

School of Engineering and Applied Sciences,
Harvard University,
Cambridge MA 02138, USA
{brchen,glpeters,mainland,mdw}@eecs.harvard.edu

Abstract. We describe *LiveNet*, a set of tools and analysis methods for reconstructing the complex behavior of a deployed sensor network. LiveNet is based on the use of multiple passive packet sniffers co-located with the network, which collect packet traces that are merged to form a global picture of the network's operation. The merged trace can be used to reconstruct critical aspects of the network's operation that cannot be observed from a single vantage point or with simple application-level instrumentation. We address several challenges: merging multiple sniffer traces, determining sniffer coverage, and inference of missing information for routing path reconstruction. We perform a detailed validation of LiveNet's accuracy and coverage using a 184-node sensor network testbed, and present results from a real-world deployment involving physiological monitoring of patients during a disaster drill. Our results show that LiveNet is able to accurately reconstruct network topology, determine bandwidth usage and routing paths, identify hot-spot nodes, and disambiguate sources of packet loss observed at the application level.

1 Introduction

As sensor networks become more sophisticated and larger in scale, better tools are needed to study their behavior in live deployment settings. Understanding the complexities of network dynamics, such as the ability of a routing protocol to react to node failure, or an application's reaction to varying external stimuli, is currently very challenging. Unfortunately, few good tools exist to observe and monitor a sensor network deployment *in situ*.

In this paper, we describe *LiveNet*, a set of tools and techniques for recording and reconstructing the complex dynamics of live sensor network deployments. LiveNet is based on the use of passive monitoring of radio packets observed from one or more *sniffers* co-deployed with the network. Sniffers record traces of all packets received on the radio channel. Traces from multiple sniffers are merged into a single trace to provide a global picture of the network's behavior. The merged trace is then subject to a series of analyses to study application behavior, data rates, network topology, routing protocol dynamics, and packet loss.

Although a passive monitoring infrastructure can increase cost, we argue that in many cases this is the ideal approach to observing and validating a sensor network's operation, and in some cases is the *only* way to effectively monitor the network. LiveNet brings a number of benefits over traditional network monitoring solutions. First, LiveNet decouples packet capture from trace analysis, allowing "raw" packet traces to be studied in many different ways. In contrast, in-network monitoring relies on preconceptions of the network's operation and failure modes, and can fail when the system does not behave as expected. Second, LiveNet requires no changes to the network being monitored, which is prudent for reasons of performance and reliability. Third, the LiveNet infrastructure can be deployed, reconfigured, and torn down separately from the network under test. LiveNet can be set up on an as-needed basis, such as during the initial sensor network deployment, or during periods when unexpected behavior is observed. Finally, it is possible to use LiveNet in situations where sensor nodes are mobile or physical access to sensor nodes is unavailable. We describe the use of LiveNet during a disaster drill involving multiple mobile patient sensor nodes, fixed repeater nodes, and base stations. In this scenario, directly instrumenting each node would have been prohibitive.

Using passive monitoring to understand a sensor network's behavior raises a number of unique challenges. First, we are concerned with the coverage of the LiveNet sniffer infrastructure in terms of total number of packets observed by the system. Second, the merging process can be affected by incomplete packet traces and lack of time synchronization across sniffers. Third, understanding global network behavior requires extracting aggregate information from the detailed traces. We describe a series of analyses, including a novel *path inference* algorithm that derives routing paths based on incomplete packet traces.

We evaluate the use of LiveNet in the context of a sensor network for monitoring patient vital signs in disaster response settings [7, 10]. We deployed LiveNet during a live disaster drill undertaken in August 2006 in which patients were monitored and triaged by emergency personnel following a simulated bus accident. We also perform an extensive validation of LiveNet using measurements on a 184-node indoor sensor network testbed.

Our results show that deploying the LiveNet infrastructure along with an existing sensor network can yield a great deal of valuable information on the network's behavior without requiring additional instrumentation or changes to the sensor network code. Our packet merging process and trace analyses yield an accurate picture of the network's operation. Finally, we show that our path inference algorithm correctly determines the routing path used without explicit information from the routing protocol stack itself.

2 Background and Motivation

Sensor networks are becoming increasingly complex, and correct behavior often involves subtle interactions between the link layer, routing protocol, and application logic. Achieving a deep understanding of network dynamics is ex-

tremely challenging for real sensor network deployments. It is often important to study a sensor deployment *in situ*, as well as in situations where it is impossible or undesirable to add additional instrumentation. Although simulators [6, 15] and testbeds [4, 17] are invaluable for development, debugging, and testing, they are fundamentally limited in their ability to capture the full complexity of radio channel characteristics, environmental stimuli, node mobility, and hardware failures that arise in real deployments. This suggests the need for *passive* and *external* observation of a sensor network's behavior "in the wild."

Several previous systems focus on monitoring and debugging live sensor deployments. Sympathy [11] is a system for reasoning about sensor node failures using information collected at *sink nodes* in the network. Sympathy has two fundamental limitations that we believe limit its applicability. First, Sympathy requires that the sensor node software be instrumented to transmit periodic *metrics* back to the sink node. However, it is often impossible or undesirable to introduce additional instrumentation into a deployed network after the fact. Second, Sympathy is limited to observe network state at sink nodes, which may be multiple routing hops from the sensor nodes being monitored. As a result, errant behavior deep in the routing tree may not be observed by the sink. However, we do believe that LiveNet could be used in conjunction with a tool like Sympathy to yield more complete information on network state.

SNMS [16] and Memento [14] are two management tools designed for inspecting state in live sensor networks. They perform functions such as neighborhood tracking, failure detection, and reporting inconsistent routing state. EnviroLog [8] is a logging tool that records function call traces to flash, which can be used after a deployment to reconstruct a node's behavior. Like Sympathy, these systems add instrumentation directly into the application code.

In contrast to these approaches, LiveNet is based on the use of passive *sniffer* nodes that capture packets transmitted by sensor nodes for later analysis. Our approach is inspired by recent work on passive monitoring for 802.11 networks, including Jigsaw [3, 2] and Wit [9]. In those systems, multiple sniffer nodes collect packet traces, which are then merged into a single trace representing the network's global behavior. A series of analyses can then be performed on the global trace, for example, understanding the behavior of the 802.11 CSMA algorithm under varying loads, or performance artifacts due to co-channel interference. Although LiveNet uses a similar trace merging approach to these systems, we are focused on a different set of challenges. In 802.11 networks, the predominant communication pattern is single hop (between clients and access points), and the focus is on understanding link level behavior. In contrast, sensor networks exhibit substantially more complex dynamics, due to multihop routing and coordinated behavior across nodes. SNIF [12] is the only other passive monitoring system for sensor networks of which we are aware. Although there are some similarities, SNIF differs from LiveNet in several important respects. SNIF is focused primarily on debugging the causes of failures in sensor networks, while we are more interested in time-varying dynamics of network behavior, such as routing path dynamics, traffic load and hotspot analysis, network connectivity,

and recovering the sources of path loss. As a result, SNIF requires less accuracy in terms of trace merging and sniffer coverage than we require with LiveNet. Also, SNIF uses sniffers that transmit complete packet traces to a base station via a Bluetooth scatternet. Apart from the scalability limitations of the scatternet itself, interference between Bluetooth and 802.15.4 radios commonly used by sensor networks is a concern, potentially causing the monitoring infrastructure to interfere with the network's operation.[1] In LiveNet, we decouple packet capture from trace analysis and forego the requirement of real-time processing, which we believe is less important for most uses of our system.

3 LiveNet Architecture

LiveNet consists of three main components: a *sniffer infrastructure* for passive monitoring and logging of radio packets; a *merging process* that normalizes multiple sniffer logs and combines them into a single trace; and a set of *analyses* that make use of the combined trace. While packet capture is performed in real time, merging and analysis of the traces is performed offline due to high storage and computational requirements. While providing real-time analysis of traffic captured by LiveNet would be useful in certain situations, we believe that offline trace merging and analysis meets many of the needs of users wishing to debug and analyze a network deployment.

3.1 Sniffer infrastructure

The first component of LiveNet is a set of passive network sniffers that capture packets and log them for later analysis. Conceptually the sniffer is very simple, consisting of a sensor node either logging packets to local flash or over its serial port to an attached host. Sniffers timestamp packets as they are received by the radio, to facilitate trace merging and timing analysis.

We envision a range of deployment options for LiveNet sniffers. Sniffers can be installed either temporarily, during initial deployment and debugging, or permanently, in order to provide an unobtrusive monitoring framework. Temporary sniffers could log packets to flash for manual retrieval, while permanent sniffers would typically require a backchannel for delivering packet logs. Another scenario might involve *mobile* sniffers, each carried by an individual around the sensor network deployment site. This would be particularly useful for capturing packets to debug a performance problem without disturbing the network configuration.

3.2 Merging process

Given a set of sniffer traces $\{S_1 \ldots S_k\}$, LiveNet's merging process combines these traces into a temporally-ordered log that represents a global view of network activity. This process must take into account the fact that each trace only

[1] The SNIF hardware is based on previous-generation radios operating in the 868 MHz band; it is unclear whether a Bluetooth scatternet backhaul could be used with current 802.15.4 radios.

contains a subset of the overall set of packets, with a varying degree of overlap. Also, we do not assume that sniffers are time synchronized, requiring that we normalize the timebase for each trace prior to merging. Finally, due to the large size of each trace, we cannot read traces into memory in their entirety. This requires a progressive approach to trace merging.

Our merging algorithm is somewhat similar to Jigsaw [3], which is designed for merging 802.11 packet traces. We briefly describe our algorithm here, referring the reader to a technical report [13] for further details. The merging algorithm operates in two phases. In the first phase, we compute a *time mapping* that maps the timebase in each trace S_i to a common (arbitrary) timebase reference S_0. Given a pair of traces S_i and S_j, we identify a unique packet p that appears in both traces. We calculate the timebase offset between S_i and S_j as the difference in the timestamps between the two packets in each trace, $\Delta i, j = t_i(p) - t_j(p)$. Δij represents a single edge in a *time graph* (where nodes are traces and edges are timebase offsets between traces). The offset $\Delta i, 0$ from a given trace S_i to the reference S_0 is then computed as the combined timebase offset along the shortest path in the time graph from S_i to S_0.

In the second phase, we progressively scan packets from each trace and apply the time correction $\Delta_{i,0}$. Packets are inserted into a priority queue ordered by global time and identical packets from across traces are merged. After filling the priority queue to a given window size W, we begin emitting packets, simply popping the top of the priority queue and writing each packet to a file. The algorithm alternates between scanning and emitting until all traces have been merged.

There are two potential sources of error in our algorithm. First, due to its progressive nature, it is possible that a packet p will be "prematurely" emitted before all instances of p in each trace have been scanned, leading to duplicates in the merged trace. In practice, we find very few duplicates using a window size $W = 10$ sec. Second, if link-layer ARQ has been used by the application, multiple retransmissions of the same packet will appear in the sniffer traces, complicating the merging process. For example, if 4 copies of a packet p appear in trace S_1, and 2 copies in trace S_2, it is not obvious how many transmissions of the packet occurred in total. We opt to adhere to the lower bound, since we know *at least* 4 transmissions of p occurred.

4 Trace Analysis

In this section, we describe a range of analysis algorithms for reconstructing a sensor network's behavior using the merged LiveNet trace. Several of these algorithms are generic and can be applied to essentially any type of traffic, while other analyses use application-specific knowledge.

4.1 Coverage analysis

The most basic analysis algorithm attempts to estimate the *coverage* of the LiveNet sniffer infrastructure, by computing the fraction of packets actually

transmitted by the network that were captured in the merged packet trace. Coverage can also be computed on a per-sniffer basis, which is useful for determining whether a given sniffer is well-placed. Let us define $C_i(n)$ as the *coverage* of sniffer S_i with respect to node n, which is simply the fraction of packets received by S_i that were actually transmitted by n. Estimating the number of packets transmitted by n can be accomplished using packet-level sequence numbers, or knowledge of the application transmission behavior (e.g., if the application transmits a periodic beacon packet). We assume that packet loss from nodes n to sniffers S_i is uniform, and does not depend on the contents of the packets. Note that this assumption might not be valid, for example, if longer packets are more likely to experience interference or path loss.

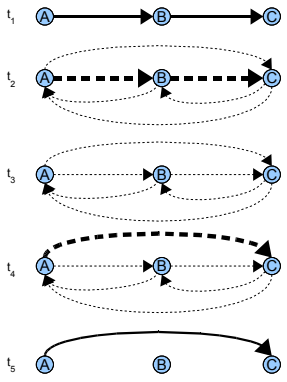
4.2 Overall traffic rate and hotspot analysis

Another basic analysis is to compute the overall amount of traffic generated by each node in the network, as well as to determine "hotspots" based on which nodes appear to be the source of, or destination of, more packets than others. Given the merged trace, we can start by counting the total number of packets originating from or destined to a given node n. Because LiveNet may not observe all actual transmissions, we would like to *infer* the existence of other packets. For example, if each transmission carries a unique sequence number we can infer missing packets by looking for gaps in the sequence number space. Coupled with topology inference (Section 4.3), one can also determine which nodes were likely to have received broadcast packets, which do not indicate their destination explicitly.

4.3 Network connectivity

Reconstructing radio connectivity between nodes is seemingly straightforward: for each packet from node a to b, we record an edge $a \rightarrow b$ in the connectivity graph. However, this approach may not reconstruct the *complete* topology, since two nodes a and b within radio range may choose not to communicate directly, depending on the routing protocol in use. We make use of two approaches. First, if one assumes that connectivity is symmetric, an edge $b \rightarrow a$ can be recorded alongside $a \rightarrow b$. Although asymmetric links are common in sensor networks [1], this algorithm would establish whether two nodes are *potential* neighbors.

The second method is to inspect routing control packets. For example, several routing protocols, such as TinyOS' `MultihopLQI`, periodically transmit their *neighbor table* containing information on which nodes are considered neighbors, sometimes along with link quality estimates. These packets can be used to reconstruct the network connectivity from the sniffer traces. Note that this information is generally not available to a base station, which would only overhear control packets within a single radio hop.



	{A,B}		{B,C}		{A,C}		{B,A}		{C,B}		{C,A}	
	Π+	Π-	Π+	Π-	Π+	Π-	Π+	Π-	Π+	Π-	Π+	Π-
t_1	**1.0**	*-0.6*	**1.0**	*-0.6*	*0.6*	**-1.0**	0	**-1.0**	0	**-1.0**	0	**-1.0**
t_2	*0.9*	*-0.7*	*0.9*	*-0.7*	*0.7*	*-0.9*	0	*-0.9*	0	*-0.9*	0	*-0.9*
t_3	*0.8*	*-0.8*	*0.8*	*-0.8*	*0.8*	*-0.8*	0	*-0.8*	0	*-0.8*	0	*-0.8*
t_4	*0.7*	*-0.9*	*0.7*	*-0.9*	*0.9*	*-0.7*	0	*-0.9*	0	*-0.9*	0	*-0.9*
t_5	*0.6*	**-1.0**	*0.6*	**-1.0**	**1.0**	*-0.6*	0	**-1.0**	0	**-1.0**	0	**-1.0**

The figure at left shows observed and inferred routing paths for a network of 3 nodes over 5 timesteps. Observed packet transmissions are shown as solid arrows; potential links are shown as dashed arrows. At time t_1, packets A→B and B→C are observed. At time t_3, packet A→C is observed. At intermediate times, the inferred path is shown in bold. At t_3, both paths A→B→C and A→C have equal probability.

The table above shows the positive (Π+) and negative (Π-) score for each link at each timestep. Values in boldface are direct observations; those in italics are time-dilated scores based on past or future values. Here we assume a time-dilation constant s = 0.1.

Fig. 1. Path inference example.

4.4 Routing path inference

One of the more interesting analyses involves reconstructing the routing path taken by a packet traveling from a source node s to a destination d. The simplest case involves protocols that use source-path routing, in which case the complete routing path is contained within the first transmission of a packet from the originating node. In most sensor network routing protocols, however, the routing state must be inferred by observing packet transmissions as packets travel from source to destination. However, because the merged packet trace may not contain every routing hop, there is some ambiguity in the routing path that is actually taken by a message. In addition, the routing path may evolve over time. As a worst case, we assume that the route can change between any two subsequent transmissions from the source node s.

The goal of our path inference algorithm is to determine the *most probable* routing path $P(s, d, t) = (s, n_1, ...n_k, d)$ at a given time t. We begin by quantizing time into fixed-sized windows; in our implementation, the window size is set to 1 sec. For each possible routing hop $a \rightarrow b$, we maintain a *score* $\Pi(a, b, t)$ that represents the likelihood of the hop being part of the routing path during the window containing t. $\Pi(a, b, t)$ is calculated using two values for each link: a *positive score* $\Pi^+(a, b, t)$ and a *negative score* $\Pi^-(a, b, t)$. The positive score represents any positive information that a link may be present in the routing path, based on an observation (possibly at a time in the past or future) that a message was transmitted from a to b. The negative score represents negative information for links that are *excluded* from the routing path due to the presence of other, conflicting links, as described below.

Figure 1 shows our algorithm at work on a simple example. We begin by initializing $\Pi^+(a, b, t) = \Pi^-(a, b, t) = 0$ for all values of a, b, and t. The merged packet trace is scanned, and for each observed packet transmission $a \rightarrow b$, we

set $\Pi^+(a,b,t) = 1$. For each *conflicting link* $a' \to b'$, we set $\Pi^-(a',b',t) = -1$. A link conflicts with $a \to b$ if it shares one endpoint in common (i.e., $a = a'$ or $b = b'$); $b \to a$ is also conflicted by definition.

Once the scan is complete, we have a sparse matrix representing the values of Π^+ and Π^- that correspond to observed packet transmissions. To fill in the rest of the matrix, we *time dilate* the scores, in effect assigning "degraded" scores to those times before and after each observation. Given a time t for which no value has been assigned for $\Pi^+(a,b,t)$, we look for the previous and next time windows $t_{prev} = t - \delta_b$ and $t_{next} = t + \delta_f$ that contain concrete observations. We then set $\Pi^+(a,b,t) = \max(\max(0, \Pi^+(a,b,t_{next}) - s \cdot \delta_f), \max(0, \Pi^+(a,b,t_{prev}) - s \cdot \delta_b))$. That is, we take the maximum value of Π^+ time-dilated backwards from t_{next} or forwards from t_{prev}, capping the value to ≥ 0. Here, s is a scaling constant that determines how quickly the score degrades per unit time; in our implementation we set $s = 0.1$. Similarly, we fill in values for missing $\Pi^-(a,b,t)$ values, also capping them to be ≤ 0.

Once we have filled in all cells of the matrix for all links and all time windows, the next step is to compute the final link score $\Pi(a,b,t)$. For this, we set the value to either $\Pi^+(a,b,t)$ or $\Pi^-(a,b,t)$ depending on which has the greater absolute value. For links for which we have no information, $\Pi(a,b,t) = 0$. The final step is to compute the most likely routing path at each moment in time. For this, we take the acyclic path that has the highest *average* score over the route, namely: $P^\star(s,d,t) = \arg\max_{\forall P(s,d,t)} \sum_{l=\{n1,n2\} \in P(s,d,t)} \Pi(n_1,n_2,t)/|P(s,d,t)|$. The choice of this metric has several implications. First, links for which we have no information ($\Pi(a,b,t) = 0$) diminish the average score over the path. Therefore, all else being equal, our algorithm will prefer shorter paths over longer ones. For example, consider a path with a "gap" between two nodes for which no observation is ever made: $(s, n_1, \ldots ? \ldots, n_2, d)$. In this case, our algorithm will fill in the gap with the direct hop $n_1 \to n_2$ since that choice maximizes the average score over any other path with more than one hop bridging the gap.

Second, note that the most likely path $P^\star$ may not be unique; it is possible that multiple routes exist with the same average score. In this case, we can use network connectivity information (Section 4.3) to exclude links that are not likely to exist in the route. While this algorithm is not guaranteed to converge on a unique solution, in practice we find that a single route tends to dominate for each time window.

4.5 Packet loss determination

Given a dynamic, multihop sensor network, one of the most challenging problems is understanding the causes of data loss from sources to sinks. The failure of a packet to arrive at a sink can be caused by packet loss along routing paths, node failures or reboots, or application-level logic (for example, a query timeout that causes a source node to stop transmitting). Using LiveNet we can disambiguate the sources of packet loss, since we can observe packet receptions from many vantage points, rather than only at the sink node.

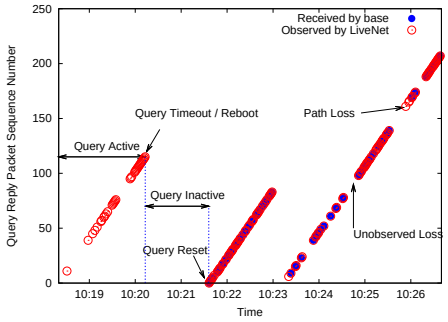


Fig. 2. Determining the causes of packet loss.

Figure 2 shows the behavior of an individual node during the disaster drill described in Section 7. Each point represents a packet either observed by LiveNet or received at the base station (or both). Here, the node is transmitting packets at 1 Hz as long as the query is active; however, the query will timeout if a renewal message is not received before it expires. Each packet carries a monotonically increasing sequence number. By combining information from the LiveNet and base station logs, we can break down packet loss in terms of loss along the routing path to the sink, inactive query periods, and *unobserved loss*; that is, packets that were neither observed by LiveNet or the sink. For example, for the first 100 sec or so, the node is transmitting packets but none of them are received by the sink, indicating a bad routing path. Query timeouts are easily detected by a reset in the packet sequence numbers. Intervals between two subsequent queries with no observed packets indicates a period with no active query.

5 Implementation

Our implementation of LiveNet consists of three components: the sniffer infrastructure, trace merging code, and analysis algorithms. The sniffers are implemented as a modified version of the TinyOS `TOSBase` application, with two important changes. First, the code is modified to pass every packet received over the radio to the serial port, regardless of destination address or AM group ID. Second, the sniffer takes a local timestamp (using the `SysTime.getTime32()` call) on each packet reception, and prepends the timestamp to the packet header before passing it to the serial port.

We observed various issues with this design that have not yet been resolved. First, it appears that TMote Sky motes have a problem streaming data at high rates to the serial port, causing packets to be dropped by the sniffer. In our LiveNet deployment described below, a laptop connected to both a MicaZ and a TMote Sky sniffer recorded more than three times as many packets from the MicaZ. This is possibly a problem with the MSP430 UART driver in TinyOS.

Second, our design only records packets received by the Active Messages layer in TinyOS. Ideally, we would like to observe control packets, such as acknowledgments, as well as packets that do not pass the AM layer CRC check.

Our merging and analysis tools are implemented in Python, using a Python back-end to the TinyOS *mig* tool to generate appropriate classes for parsing the raw packet data. A parsing script first scans each raw packet trace and emits a parsed log file in which each packet is represented as a *dictionary* mapping named keys to values. Each key represents a separate field in the packet (source ID, sequence number, and so forth). The dictionary provides an extremely flexible mechanism for reading and manipulating packet logs, simplifying the design of the merging and subsequent analysis tools. The merging code is 657 lines of code (including all comments). The various analysis tools comprise 3662 lines of code in total. A separate library (131 lines of code) is used for parsing and managing packet traces, which is shared by all of the merging and analysis tools.

6 Validation Study

The goal of our validation study is to ascertain the accuracy of the LiveNet approach to monitoring and reconstructing sensor network behavior. For this purpose, we make use of a well-provisioned indoor testbed, which allows us to study LiveNet in a controlled setting. The MoteLab [17] testbed consists of 184 TMote Sky nodes deployed over three floors of the Harvard EECS building, located mainly on bookshelves in various offices and labs. During the experiments between 120–130 nodes were active. Each node is connected to a USB-Ethernet bridge for programming and access to the node's serial port. For our validation, half of the nodes are used as sniffers and the other half used to run various applications. Although such a sniffer ratio is much larger than we would expect in a live deployment, this allows us to study the effect of varying sniffer coverage.

6.1 Sniffer reception rate

The first consideration is how well a single sniffer can capture packets at varying traffic rates. For these experiments, we make use of a simple TinyOS application that periodically transmits packets containing the sending node ID and a unique sequence number. Figure 3 shows the reception rate of two sniffers (a MicaZ and a TMote Sky) with up to 4 nodes transmitting at increasing rates. All nodes were located within several meters of each other. Note that due to CSMA backoff, the offered load may be lower than the sum of the transmitter's individual packet rates. We determine the offered load by computing a linear regression on the observed packet reception times at the sniffer.

As the figure shows, a single sniffer is able to sustain an offered load of 100 packets/sec, after which reception probability degrades. Note that the default MAC used in TinyOS limits the transmission rate of short packets to 284 packets/sec. Also, as mentioned in Section 5, MicaZ-based sniffers can handle somewhat higher loads than the TMote Sky. We surmise this to be due to differences in the serial I/O stack between the two mote platforms.

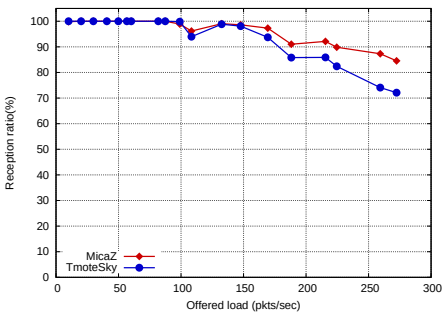


Fig. 3. Sniffer reception rate vs. offered load.

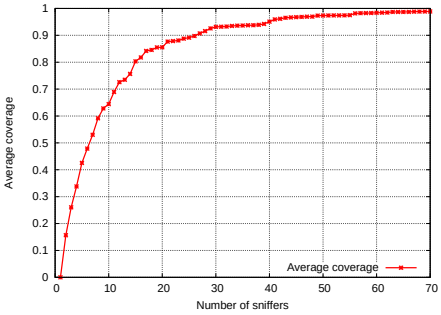


Fig. 4. Sniffer coverage.

6.2 Merge performance

Although LiveNet's sniffer traces are intended for offline analysis, the performance of the trace merging process is potentially of interest. For this experiment, we merge up to 25 traces containing 1000 sec of packet data on an unloaded 2.4 GHz Linux desktop with 1 GB of memory. Merging two traces takes 301 sec, 10 traces 1418 sec, and 25 traces 2859 sec. The "break even" point where merging time takes longer than the trace duration occurs at about 8-10 traces. This suggests that for a modest number of traces, one could conceivably perform merging in real time, although this was not one of our design goals. Note that we have made no attempt to optimize the LiveNet merge code, which is implemented in Python and makes heavy use of ASCII files and regular expression matching.

6.3 Coverage

The next question is how many sniffers are required to achieve a given *coverage* in our testbed. We define coverage as the fraction of transmitted packets that are received by the LiveNet infrastructure. There are 70 sniffer traces in total for this experiment.

To compute the coverage of a random set of N traces, the most thorough, yet computationally demanding, approach is to take all $\binom{70}{N}$ subsets of traces, individually merge each subset, and compute the resulting coverage. Performing this calculation would be prohibitively expensive. Instead, we estimate the coverage of N traces by taking multiple random permutations of the traces and successively merge them, adding one trace at a time to the merge and computing the resulting coverage, as described below.

Let S_i represent a single trace and $\mathcal{S} = \{S_1 \ldots S_{70}\}$ represent the complete set of traces. Let $M(S_1 \ldots S_k)$ represent the merge of traces $S_1 \ldots S_k$, and $C(k)$ represent the coverage of these k merged traces, as defined in Section 4.1. We start by computing the coverage of the first trace S_1, yielding $C(1)$. We then merge the first two traces $M(S_1, S_2)$ and compute the coverage $C(2)$ of this

merge. Next, we successively add one trace at a time to the merge, computing the resulting coverage for each trace until we have computed $C(1) \ldots C(70)$.

To avoid sensitivity to the order in which the original traces are numbered, we generate five random permutations $\mathcal{S}'$ of the original traces and compute the coverage $C(k)$ accordingly. Our final estimate of the coverage of k traces is the average of the five values of $C(k)$ computed for each of the permutations. The results are shown in Figure 4.

As the figure shows, the first 17 traces yield the greatest contribution, achieving a coverage of 84%. After this, additional sniffers result in diminishing returns. A coverage of 90% is reached with 27 sniffers, and all 70 sniffers have a coverage of just under 99%. Of course, these results are highly dependent on the physical extent and placement of our testbed nodes. The testbed covers 3 floors of a building spanning an area of 5226m^2. Assuming nodes are uniformly distributed in this area (which is not the case), this suggests that approximately one sniffer per 193m^2 would achieve a coverage of 90%. Keep in mind that sniffer locations were not planned to maximize coverage, and we are using the built-in antenna of the TMote Sky. High-gain antennas and careful placement would likely achieve better coverage with fewer nodes.

6.4 Merge accuracy

Next, we are interested in evaluating the accuracy of the merged trace. As described earlier, our trace merging algorithm operates on fixed-length time windows and could lead to duplicate or reordered packets in the merged trace. After merging all 70 source traces from the previous experiment, we captured a total of 246,532 packets. 2920 packets are missing from the trace (coverage of 98.8%). There are a total of 354 duplicate packets (0.14%), and 13 out-of-order packets (0.005%). We feel confident that these error rates are low enough to rely on the merged trace for higher-level analyses.

6.5 Path inference

To test the path inference algorithm described in Section 4.4, we set up an experiment in which one node routes data to a given sink node over several multihop paths. The node is programmed to automatically select a new route to the sink every 5 minutes. Since we know the routing paths in advance, we can compare the path inference algorithm against ground truth.

Space limitations prevent us from presenting complete results here, though we refer the reader to our technical report [13] for more details. In summary, the path inference algorithm correctly determined the routing path chosen by the network in all cases. When the routing path changed, the algorithm would incorrectly determine that an "intermediate" route was being used, but this would occur for no more than 1 or 2 sec until the correct route was observed.

Fig. 5. The indoor treatment area of the disaster drill. *Inset shows the electronic triage tag.*

7 Deployment Study: Disaster Drill

To evaluate LiveNet in a realistic application setting, we deployed the system as part of a live disaster drill that took place in August 2006 in Baltimore, MD, in collaboration with the AID-N team at Johns Hopkins Applied Physics Laboratory and rescue workers from Montgomery County Fire and Rescue Services. Disaster response and emergency medicine offer an exciting opportunity for use of wireless sensor networks in a highly dynamic and time-critical environment. Understanding the behavior of this network during a live deployment is essential for resolving bugs and performance issues.

The disaster drill modeled a simulated bus accident in which twenty volunteer "victims" were triaged and treated on the scene by 13 medics and firefighters participating in the drill. Each patient was outfitted with one or more sensor nodes to monitor vital signs, which formed an *ad hoc* network, relaying real-time data back to multiple laptop base stations located at the incident command post nearby. Each laptop displayed the triage status and vital signs for each patient, and logged all received data to a file. The incident commander could rapidly observe whether a given patient required immediate attention, as well as update the status of each patient, for example, by setting the triage status from "moderate" to "severe."

The network consisted of two types of sensor nodes: an *electronic triage tag* and a *electrocardiograph* (ECG) [5, 10]. The triage tag incorporates a pulse oximeter (monitoring heart rate and blood oxygen saturation using a small sensor attached to the patient's finger), an LCD display for displaying vital signs, and multiple LEDs for indicating the patient's triage status (green, yellow, or red, depending on the patient's severity). The triage tags are based on the MicaZ mote with a custom daughterboard and case. The ECG node consists of a TMote Sky with a custom sensor board providing a two-lead (single-channel)

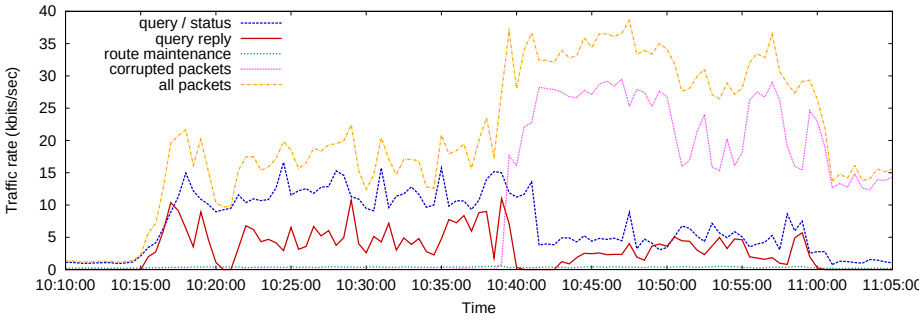


Fig. 6. Overall traffic rate during the disaster drill.

electrocardiograph signal. In addition to the patient sensor nodes, a number of static repeater nodes were deployed to assist with maintaining network connectivity. The sensor nodes and repeaters all ran the CodeBlue system [7], which is designed to support real-time vital sign monitoring and triage for disaster response.

Our goal in deploying LiveNet was to capture detailed data on the operation of the network as nodes were activated, patients moved from the triage to treatment areas, and study the scalability and robustness of our *ad hoc* networking protocols. In this situation, it would have been impossible to record complete packet traces from each sensor node directly, motivating the need for a passive monitoring infrastructure. We made use of 6 separate sniffer nodes attached to 3 laptops (the laptops had two sniffers to improve coverage).

Figure 5 shows a picture from the drill to give a sense of the setup. The drill occurred in three stages. The first stage occurred in a parking lot area outdoors during which patients were outfitted with sensors and initial triage performed. In the second stage, most of the patients were moved to an indoor treatment area as shown in the picture. In the third stage, two of the "critical" patients were transported to a nearby hospital. LiveNet sniffers were placed in all three locations. Our analysis in this paper focuses on data from 6 sniffers located at the disaster site. The drill ran for a total of 53 minutes, during which we recorded a total of 110548 packets in the merged trace from a total of 20 nodes (11 patient sensors, 6 repeaters and 3 base stations).

8 Deployment evaluation

In this section, we perform an evaluation of the LiveNet traces gathered during the disaster drill described in Section 7.

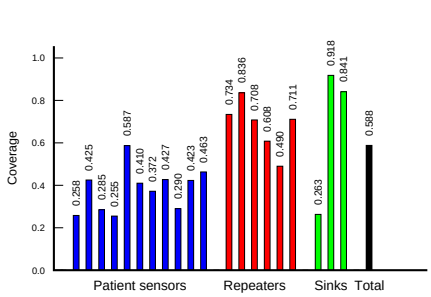


Fig. 7. Per-node coverage during the drill.

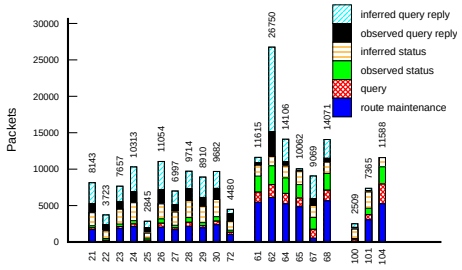


Fig. 8. Inferred per-node packet traffic load during the drill.

8.1 General evaluation

As a general evaluation of the sensor network's operation during the drill, we first present the overall traffic rate and packet type breakdown in Figure 6 and 10. These high-level analyses help us understand the operation of the deployed network and can be used to discover performance anomalies that are not observable from the network sinks.

As Figure 6 shows, at around time $t = 10 : 39$ there is a sudden increase in corrupted packets received by LiveNet: these packets have one or more fields that appear to contain bogus data. Looking more closely at Figure 10, starting at this time we see a large number of partially-corrupted routing protocol control messages being flooded into the network. On closer inspection, we found that these packets were otherwise normal spanning-tree maintenance messages that contained bogus sequence numbers. This caused the duplicate suppression algorithm in the routing protocol to fail, initiating a perpetual broadcast storm that lasted for the entire second half of the drill. The storm also appears to have negatively affected application data traffic as seen in Figure 6.

We believe the cause to be a bug in the routing protocol (that we have since fixed) that only occurs under heavy load. Note that we had no way of observing this bug without LiveNet, since the base stations would drop these bogus packets.

8.2 Coverage

To determine sniffer coverage, we make use of periodic status messages broadcast by each sensor node once every 15 sec. Each status message contains the node ID, sensor types attached, and a unique sequence number. The sequence numbers allow us to identify gaps in the packet traces captured by LiveNet, assuming that all status messages were in fact transmitted by the node.

Figure 7 shows the coverage broken down by each of the 20 nodes in the disaster drill. There were a total of 4819 expected status messages during the

run, and LiveNet captured 59% overall. We observed 89 duplicate and out-of-order packets out of 2924 packets in total, for an error rate of 3%. As the figure shows, the coverage for the fixed repeater nodes is generally greater than for the patient sensors; this is not too surprising as the patients were moving between different locations during the drill, and several patients were lying on the ground. The low coverage (26%) for one of the sink nodes is because this node was located inside an ambulance, far from the rest of the deployment.

8.3 Topology and network hotspots

The network topology during the drill was very chaotic, since nodes were moving and several nodes experienced reboots. Such a complex dataset is too dense to show as a figure here. However, we can discuss a few observations from analyzing the topology data. First, most nodes are observed to use several outbound links, indicating a fair amount of route adaptation. There are multiple routing trees (one rooted at each of the sinks), and node mobility causes path changes over time. Second, all but two of the patient sensors have both incoming and outgoing unicast links, indicating that patient sensors performed packet relaying for other nodes. Indeed, one of the sink nodes also relayed packets during the drill. We also observe that one of the patient sensors transmitted unicast packets to itself, suggesting corruption of the routing table state on that node; this is a protocol bug that would not have been observed without LiveNet.

Besides topology, it is useful to identify "hotspots" in the network by computing the total number of packets transmitted by each node during the drill. For several message types (query reply and status messages), we can also infer the existence of unobserved packets using sequence numbers; this information is not available for route maintenance and query messages. Figure 8 shows the breakdown by node ID and packet type. Since we are counting all transmissions by a node, these totals include forwarded packets, which explains the large variation from node to node.

The graph reveals a few interesting features. Repeater nodes generated a larger amount of traffic overall than the patient sensors, indicating that they were used heavily during the drill. Second, node 62 (a repeater) seems to have a very large number of inferred (that is, unobserved) query reply packets, far more than its coverage of 83.6% would predict. This suggests that the node internally dropped packets that it was forwarding for other nodes, possibly due to heavy load. Note that with the information in the trace, there is no way to disambiguate packets unobserved by LiveNet from those dropped by a node. If we assume that our coverage calculation is correct, we would infer a total of 4108 packets; instead we infer a total of 15017. Node 62 may have then dropped as many as $(15017-4108)/15017 = 72\%$ of the query replies it was forwarding. With no further information, the totals in Figure 8 therefore represent a conservative upper bound on the total traffic transmitted by the node.

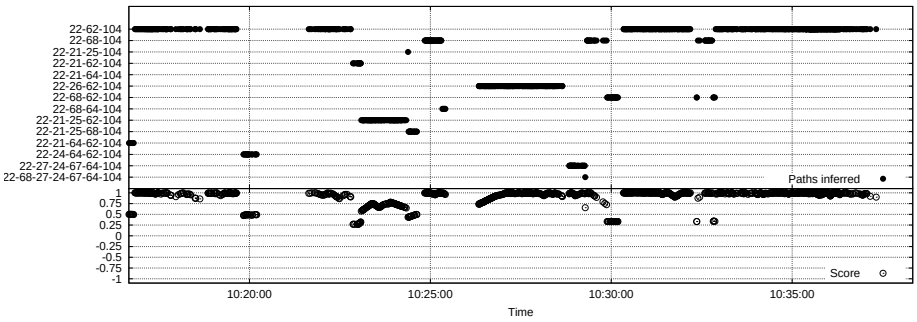


Fig. 9. Routing paths for one of the patient sensors during the disaster drill. *The upper graph shows the inferred routing path over time, while the lower graph shows the Π score for the most probable path.*

8.4 Path inference

Given the limited coverage of the LiveNet sniffers during the drill, we would not expect the path inference algorithm to provide results as clear as those in our testbed. As shown in Figure 9, we take a single patient sensor (node 22) and infer the routing paths to one of the sink nodes (node 104). The results are shown in Figure 9. As the figure shows, there are many inferred paths that lack complete observations of every hop, and the Π scores for these paths vary greatly. The path $22 \rightarrow 62 \rightarrow 104$ is most common; node 62 is one of the repeaters. In several cases the routing path alternates between repeaters 62 and 68. The node also routes packets through other repeaters and several other patient sensors. The longest inferred path is 5 hops.

This example highlights the value of LiveNet in understanding fairly complex communication dynamics during a real deployment. We were surprised to see such frequent path changes and so many routing paths in use, even for a single source-sink pair. This data can help us tune our routing protocol to better handle mobility and varying link quality.

8.5 Query behavior and yield

The final analysis that we perform involves understanding the causes of data loss from sources to sinks. We found that the overall data yield during the drill was very low, with only about 20% of the expected data transmitted by patient sensors reaching the sink nodes. There are three distinct sources of data loss: (1) packet loss along routing paths; (2) node failures or reboots, and (3) query timeouts.

During the drill, most patient nodes ran six simultaneous queries, sending two types of vital sign data at 1 Hz to each of the three sinks. Each data packet carries a unique sequence number. The CodeBlue application uses a lease model

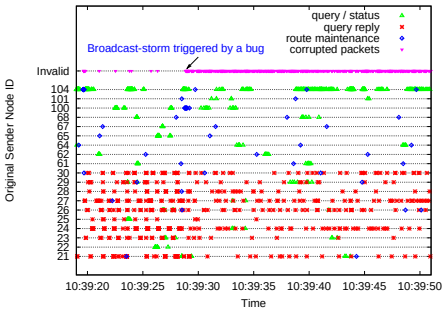


Fig. 10. Packet type breakdown for a portion of the disaster drill.

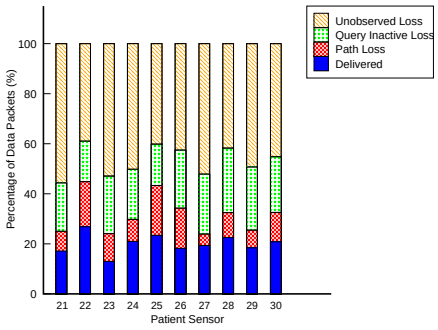


Fig. 11. Query yield per node during the disaster drill.

for queries, in which the node stops transmitting data after a timeout period unless the lease is renewed by the sink. Each sink was programmed to re-issue the query 10 seconds before the lease expires; however, if this query is not received by the node in time, the query will time out until the sink re-issues the query.

Using the analysis described in Section 4.5, we can disambiguate the causes of packet loss. Figure 11 shows the results. A query yield of 100% corresponds to the sink receiving packets at exactly 1 Hz during all times that the node was alive. As the figure shows, the actual query yield was 17–26%. Using the LiveNet traces, we can attribute 5–20% loss to dropped packets along routing paths, and 16–25% loss to premature query timeouts. The rest of the loss is unobserved, but likely corresponds to path loss since these packets should have been transmitted during query active periods.

This analysis underscores the value of the LiveNet monitoring infrastructure during the deployment. Without LiveNet, we would have little information to help us tease apart these different effects, since we could only observe those packets received at the sinks. With LiveNet, however, we can observe the network's operation in much greater detail. In this case, we see that the query timeout mechanism performed poorly and needs to be made more robust. Also, routing path loss appears to be fairly high, suggesting the need for better reliability mechanisms, such as FEC.

9 Future Work and Conclusions

LiveNet provides an infrastructure for monitoring and debugging live sensor network deployments. Rather than introducing possibly intrusive instrumentation into the sensor application itself, we rely on passive, external packet monitoring coupled with trace merging and high-level analysis. We have shown that LiveNet is capable of scaling to a large number of sniffers; that merging is very accurate

with respect to ground truth; that a small number of sniffers are needed to achieve good trace coverage; and that LiveNet allows one to reconstruct the behavior of a sensor network in terms of traffic load, network topology, and routing paths. We have found LiveNet to be invaluable in understanding the behavior of the disaster drill deployment, and intend to leverage the system for future deployments as well.

LiveNet is currently targeted at online data collection with offline post processing and analysis. We believe it would be valuable to investigate a somewhat different model, in which the infrastructure passively monitors network traffic and alerts an end-user when certain conditions are met. For example, LiveNet nodes could be seeded with event detectors to trigger on interesting behavior: for example, routing loops, packet floods, or corrupt packets. To increase scalability and decrease merge overhead, it may be possible to perform *partial merging* of packet traces around windows containing interesting activity. In this way the LiveNet infrastructure can discard (or perhaps summarize) the majority of the traffic that is deemed uninteresting.

Such an approach would require decomposing event detection and analysis code into components that run on individual sniffers and those that run on a backend server for merging and analysis. For example, it may be possible to perform merging in a distributed fashion, merging traces pairwise along a tree; however, this requires that at each level there is enough correspondence between peer traces to ensure timing correction can be performed. Having an expressive language for specifying trigger conditions and high-level analyses strikes us as an interesting area for future work.

Acknowledgments

The authors gratefully acknowledge the contributions of the AID-N team at Johns Hopkins Applied Physics Laboratory for their assistance with the disaster drill logistics as well as essential hardware and software development: Tia Gao, Tammara Massey, Dan Greenspan, Alex Alm, Jonathan Sharp, and David White. Leo Selavo (Univ. Virginia) developed the wireless triage tag used in the drill. Konrad Lorincz and Victor Shnayder (Harvard) contributed to the CodeBlue software platform. Geoff Werner-Allen (Harvard) developed the MoteLab sensor network testbed used for our validation studies.

References

1. N. B. Alberto Cerpa and D. Estrin. Scale: a tool for simple connectivity assessment in lossy environments. Technical Report CENS TR-0021, UCLA, September 2003.
2. Y.-C. Cheng, M. Afanasyev, P. Verkaik, P. Benko, J. Chiang, A. C. Snoeren, S. Savage, , and G. M. Voelker. Automating cross-layer diagnosis of enterprise wireless networks. In *ACM SIGCOMM Conference*, 2007.
3. Y.-C. Cheng, J. Bellardo, P. Benko, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *ACM SIGCOMM Conference*, 2006.

4. B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, , and A. Vahdat. Mirage: A Microeconomic Resource Allocation System for SensorNet Testbeds. In *Proc. the the Second IEEE Workshop on Embedded Networked Sensors (EMNETS'05)*, May 2005.
5. T. R. F. Fulford-Jones, G.-Y. Wei, and M. Welsh. A portable, low-power, wireless two-lead ekg system. In *Proc. the 26th IEEE EMBS Annual International Conference*, San Francisco, September 2004.
6. P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.
7. K. Lorincz, D. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, S. Moulton, and M. Welsh. Sensor Networks for Emergency Response: Challenges and Opportunities. *IEEE Pervasive Computing*, Oct-Dec 2004.
8. L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, , and J. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Proc. IEEE INFOCOM Conference*, Barcelona, Spain, April 2006.
9. R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing the mac-level behavior of wireless networks in the wild. In *ACM SIGCOMM Conference*, 2006.
10. T. Massey, T. Gao, M. Welsh, , and J. Sharp. Design of a decentralized electronic triage system. In *Proc. American Medical Informatics Association Annual Conference (AMIA 2006)*, Washington, DC, November 2006.
11. N. Ramanathan, K. Chang, L. Girod, R. Kapur, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3nd international conference on Embedded networked sensor systems*, 2005.
12. M. Ringwald, K. Römer, and A. Vitaletti. Passive inspection of sensor networks. In *Proceedings of the 3rd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2007)*, Santa Fe, New Mexico, USA, June 2007.
13. B. rong Chen, G. Peterson, G. Mainland, and M. Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. Technical Report TR-11-07, School of Engineering and Applied Sciences, Harvard University, Aug. 2007.
14. S. Rost and H. Balakrishnan. Memento: A Health Monitoring System for Wireless Sensor Networks. In *IEEE SECON*, Reston, VA, September 2006.
15. B. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proc. Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.
16. G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proc. the 2nd European Confererence on Wireless Sensor Networks (EWSN 2005)*, Jan. 2005.
17. G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A Wireless Sensor Network Testbed. In *Proc. the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, April 2005.