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
{brchen,glpeters,mainland,mdw}@eecs.harvard.edu

Abstract

Understanding the behavior of deployed sensor networks is difficult as they become more sophisticated and larger in scale. Much of the difficulty comes from the lack of tools to provide a global view on the network dynamics. This paper describes *LiveNet*, a set of tools and techniques for reconstructing complex dynamics of live sensor network deployments. LiveNet is based on the use of passive *sniffers* co-deployed with the network. We address several challenges: merging multiple sniffer traces, determining coverage of sniffers, inference of missing information for path reconstruction and high-level analyses with application-specific knowledge. To validate LiveNet's accuracy, we conduct controlled experiments on an indoor testbed. Finally, we present data from a real deployment using LiveNet. The results show that LiveNet is able to to reconstruct network topology, bandwidth usage, routing paths, identify hot-spot nodes, and disambiguate failures observed at application level without instrumenting application code.

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. Simulators [7, 15] and testbeds [17, 3] offer the opportunity to understand sensor applications in controlled settings. However, no good tools exist to observe and monitor a sensor network deployment *in situ*.

In many cases, it is difficult or impossible to add new instrumentation into a deployed sensor network. Depending on the circumstances, reprogramming nodes may be impossible or inadvisable (for example, due to downtime or risk of breaking an existing system). Also, adding debugging code adds overhead and consumes memory, CPU, and network resources. Additionally, the instrumentation code itself may alter the behavior of the deployed network in subtle ways. Using a wired backchannel (as is commonly used in testbeds) is not possible in deployments where nodes may be mobile or located in remote environments.

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. Sniffer nodes can be either temporary or permanent, fixed or mobile, and wired or untethered. Sniffers record traces of all packet activity observed 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, and routing protocol dynamics.

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. We make use of an approach similar to Jigsaw [2] and Wit [9] to merge multiple traces, with modifications specific to the use of 802.15.4 networks. 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. We deployed LiveNet during a live disaster drill undertaken in August 2006 in which 10 patients were monitored and triaged following a simulated bus accident. During the deployment, the network experienced highly variable performance due to node mobility and a bug (only discovered using the LiveNet traces) causing massive packet flooding. We also perform an extensive validation of LiveNet using measurements on an indoor sensor network testbed.

Our results show that deploying a 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.

The rest of this paper is organized as follows. In Section 2, we provide motivation and background for the LiveNet approach, and discuss related work. Section 3 describes the LiveNet architecture, and Section 4 discusses a range of high-level analyses that can be performed on the merged sniffer traces. Implementation details are given in Section 5, and we give an overview of the disaster drill deployment in Section 6. In Section 7 we validate the use of LiveNet on an indoor testbed in a controlled setting, and in Section 8 we evaluate the use of LiveNet on the disaster drill dataset. Finally, Section 9 discusses future work and concludes.

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 extremely challenging for real sensor network deployments. It is often important to study a sensor deployment *in situ*, that is, in its "natural" setting (rather than as part of a testbed or simulation), as well as in situations where it is impossible or undesirable to add additional instrumentation. These requirements suggest the need for *passive* and *external* observation of sensor network behavior, and is the goal of our work.

The most common approach to sensor network development is simulation [7, 14, 15], which provides an easy means for instrumenting code and observing the global behavior of the (simulated) application. However, the behavior of a deployed network may vary substantially from simulation results. Although simulation is an invaluable development and debugging tool, no simulator can perfectly capture the environment, radio channel characteristics, variations in hardware calibration, and other effects that are so vexing for real world deployments.

Sensor network testbeds [3, 17, 5, 18] provide a more realistic debugging environment. However, testbeds are generally deployed in controlled settings (such as office buildings or laboratories) and make use of wired backchannels for powering, programming, and communicating with individual nodes. In fielded sensor networks, however, such an approach is clearly impractical. This problem is compounded in applications involving mobile sensor nodes.

Sensor network debugging tools: Several previous systems focus on monitoring and debugging live sensor deployments. Sympathy [10] 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. In many cases, it is often impossible or undesirable to introduce additional instrumentation into a live deployment. It may not be possible to reprogram sensor nodes following a deployment (especially if the nodes are closed or provided by a third party), and instrumentation can change the behavior of the network in subtle ways. Second, Sympathy is limited to observe network state at sink nodes which may be multiple routing hops from the sensor nodes in question. As a result, errant behavior deep in the routing tree may not be observed by the sink. We 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 [13] 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. Like Sympathy, these systems involve adding code to the sensor network application. Both systems attempt to minimize the amount of information they transmit to limit bandwidth and energy consumption. EnviroLog [8] is a logging tool that is compiled into sensor network applications. It records function call traces to the node's flash. After deployment, EnviroLog can replay the call trace to replicate node behavior.

Passive monitoring tools: LiveNet provides a framework that permits passive monitoring of a sensor network *in situ*, based on the use of one or more *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, namely Jigsaw [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 cochannel interference.

Several other papers have leveraged passive monitoring for studying 802.11 networks. Jigsaw and Wit build upon earlier work by Yeo *et al.* [19] for trace merging. Jardosh *et al.* [6] and Rodrig *et al.* [12] describe trace-based analysis of 802.11 behavior in congested settings.

Passive monitoring requires that we merge traces from multiple sniffers into a single trace, which is made difficult by varying packet loss between nodes and sniffers as well as the lack of a global timebase. The key idea is to identify unique packets in each trace and first time-correct traces to determine their correspondence. Although LiveNet uses a similar trace merging approach to Jigsaw and Wit (described in Section 3.2), leveraging this approach for sensor networks raises new challenges not seen in typical 802.11 settings.

The first challenge is dealing with heterogeneity in the sniffer infrastructure. One example is that that each sniffer might record slightly different information for the *same* packet transmitted over the radio: for example, the MicaZ and TMote Sky nodes use different packet formats when relaying information to their serial port.

Another set of challenges involves understanding the network's global operation from a set of passively acquired packet traces, with no additional instrumentation. Since sensor networks generally involve multihop communication, this task is more difficult than in 802.11 networks with single-hop communication between clients and access points. For example, Jigsaw and Wit focus on link-layer and client-to-AP traffic, while we are concerned with network-wide dynamics.

We must also be concerned with the *coverage* of the monitoring infrastructure, and whether there are any "blind spots" caused by poorly-placed sniffers or packet loss due to interference, which may lead to drawing incorrect conclusions about the network's operation. In Section 4.4 we describe our algorithm for reconstructing routing paths from this incomplete information.

The most similar approach to LiveNet is SNIF [11], which is also based on passive monitoring. However, SNIF focuses on interactive debugging and transient failure detection, while LiveNet aims to reconstruct sensor network behavior over longer periods of operation. Few published details are available on SNIF at the time of writing, so it is unclear what approach the system uses for merging and analyzing packet traces. We believe that the SNIF and LiveNet approaches complement each other as our goals appear to be fairly distinct.

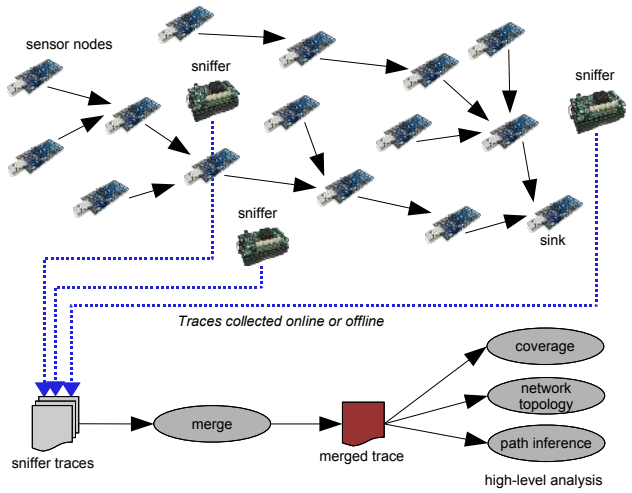


Figure 1: **Overview of the LiveNet architecture.**

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 core of the LiveNet infrastructure 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. However, certain factors must be taken into account to achieve the desired level of fidelity in the recorded packet trace.

First of all, sniffers must timestamp packets as they are received by the radio, to facilitate later timing-based analysis as well as merging of traces. This timestamping can be performed at different levels depending on the desired degree of accuracy. Stamping packets as they are received by the mote's radio stack offers the lowest latency between packet reception and timestamp capture. Timestamping as the packet is received by the sniffer application may suffer delays due to packet processing (e.g., posting the packet receive task in TinyOS). Timestamping at the host attached to the sniffer, if any, involves the highest latency as packets must be transferred over the serial port prior to stamping.

Our prototype uses hardware timers provided by the CPU (as appropriate for each mote platform) to timestamp packets in the *ReceiveMsg.receive()* interface; on the MSP430 this is a 32 kHz timer. Hooking into the CC2420 radio stack would offer lower latency for timestamping, however, none of the analyses that we perform required this higher degree of accuracy.

We assume that all sensor nodes are sharing the same radio channel and that sniffers are tuned to the same channel as the network. In cases where frequency hopping is used, sniffers need to *scan* multiple channels or match the frequency-hopping schedule of the sensor nodes. However, frequency hopping is not common in 802.15.4-based sensor networks deployed to date.

In our current prototype, each received radio packet is wrapped in an envelope and enqueued for transmission over the mote's serial port. The envelope contains the timestamp, ID of the sniffer node, and original radio packet contents. A host attached to the mote is responsible for logging the packet. This host could be a laptop or PC, or a USB-to-Ethernet bridge such as the TMote Connect. The host logs each packet in a raw format as it is received over the serial port to a file or database.

Deployment scenarios: 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 one or more *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 $\{S_1 \ldots S_k\}$, a core problem is how to combine these traces into a temporally-ordered log that represents the union of the packets observed by each sniffer, offering a global view of network activity.

Several challenges arise when merging multiple sniffer traces. First, each trace will contain only a subset of the overall set of packets received, so it is important to interleave multiple traces while preserving the correct packet order. Second, we do not assume that sniffers are time synchronized, so two sniffers logging the same packet will use different timestamps. Therefore, we must first *normalize* the timebase used across each trace, in effect performing *post hoc* time synchronization. Third, link-level retransmission of a packet can cause multiple copies of the same packet contents to appear in each sniffer trace, making it difficult to disambiguate individual packet transmissions from each other.

Our approach to trace merging is inspired by Jigsaw [2] and Wit [9], which perform merging of multiple 802.11 packet traces. Although there are similarities between these systems and LiveNet, as described in Section 2, there are important differences owing to the nature of sensor networks, in particular multihop traffic. We briefly describe the process here, referring the reader to [2, 9] for additional background.

3.2.1 Timing normalization

The first step is to *time correct* each of the sniffer traces so that a unique packet logged in multiple traces will have a (near) identical timestamp. We choose an arbitrary sniffer trace S_0 as a timebase reference and map packets in all other traces to this timebase. Let t_p^i represent the timestamp associated with packet p logged in trace S_i. Our goal is to assign a timestamp $\hat{t}_p^i$ such that $\hat{t}_p^i \approx t_p^0$ (assuming that p was logged in sniffer trace S_0). Note that due to slight timing jitter between sniffers, even after time correction, not all instances of p logged in all traces will have an identical corrected timestamp $\hat{t}_p^i$.

To accomplish this, we calculate a *time mapping* Δ_i such that $\hat{t}_p^i = t_p^0 + \Delta_i$. The time mapping represents a constant time offset between packets logged in S_0 and S_i. This accounts for a constant time shift between two sniffers. Rather than directly account for clock skew, we partition the sniffer trace into multiple *intervals* and compute Δ_i separately for each interval. The duration of the interval is selected so that any accumulated clock skew will not introduce large errors in the corrected timestamps; we set the interval length to 1 hour in our prototype.

In the case where sniffers S_0 and S_i both log the same packet p, computing the time mapping Δ_i is straightforward: $\Delta_i = t_p^i - t_p^0$. However, depending on the physical placement of sniffers, it is likely that S_i and S_0 may not overhear any packets in common. In general, we compute the time mapping using a *time offset graph* $G(V, E)$ where each vertex in V represents a sniffer, and edges in E represent the time offset between two sniffers that log at least one packet p in common. The weight of each edge $E(S_i, S_j) = \Delta_{i,j}$. The time offset $\Delta_i = \Delta_{i,0}$ is computed by taking the shortest weighted path in the time graph from S_i to S_0.

The time mapping algorithm proceeds as follows. Each trace is scanned one packet at a time and a hash of the contents of the packet $h(p)$ is computed. We record the tuple (S_i, t_p^i) in a hashtable with key $h(p)$. If the same packet is recorded in multiple traces, these entries will match the same hash bucket. When this occurs, we add two edges to the time graph with weights $\Delta_{i,j}$ and $\Delta_{j,i}$ respectively. If Δ_i or Δ_j are unknown, we test for a shortest path to S_0 in the time graph to compute the missing value. The algorithm terminates when either Δ_i has been computed for all sniffers i, or all packets from all sniffer logs have been read. It is possible that after this process we cannot compute Δ_i for certain sniffers, at which point those sniffers are excluded from further analysis.

To avoid ambiguity in computing the time mapping, we only consider *unique* packets p in the sniffer traces, that is, packets that will not be logged multiple times by the same sniffer with multiple timestamps. For example, when using link-layer ARQ, a packet might be transmitted multiple times with an identical payload, leading to an ambiguity about which instances of p correspond to each other in different sniffer traces. For timebase construction, we only consider packets that are expected to be unique in a given sniffer trace: for example, any packet with a unique sequence number assigned by the transmitting node. Rollover in the sequence numbers (for example, if only an 8-bit sequence number field is used) is easily handled by first pre-scanning the sniffer traces and assigning a *pseudo-sequence number* that will not roll over (e.g., using a 64-bit value).

3.2.2 Progressive trace merging

The second step is to merge the sniffer traces and write out a single combined trace. Because individual traces can be very large, it is impractical to read each trace into memory, perform a merge, and write out the result. For example, traces from the disaster drill deployment (described in Section 8) were up to 37 MB in size for a log covering less than 90 minutes of trace data. This requires that we perform a *progressive* merging of individual traces, writing out the merged results on the fly.

Our algorithm operates by alternating between two phases: *scanning* and *emitting*. In the scanning phase, we scan packets from each trace S_i. The packet contents are read and the time mapping Δ_i applied to each packet. Packets are inserted into a priority queue sorted by timestamp. If a packet with identical contents (as determined by the hash function $h(p)$) is already in the priority queue, it is *merged* into a single packet, retaining information about the set of traces in which the packet was identified and the maximum time spread $\Delta t = \max_{\forall S_i, S_j}(\|t_p^i - t_p^j\|)$ between any two instances of the packet in the priority queue. We continue scanning packets from the same source S_i until $\hat{t}_p^i$ is greater than or equal to the highest timestamp of packets already in the priority queue.

Once we have scanned all traces $\{S_1 \ldots S_k\}$ in this manner, we begin an emitting phase. We iterate over the priority queue (ordered by the time-corrected timestamp) and emit each packet p into the merged trace. (Recall that p may have been merged with identical packets from other traces during the scanning phase.) The emitting phase continues until one of two conditions occurs: (1) The packet p popped from the priority queue has a time offset of σ sec greater than the previously emitted packet, or (2) The length of the priority queue (time offset between the first and last packet) is less than η sec. In the former case, we withhold emitting the packet, place it back on the priority queue and, return to the scanning phase. The idea is that if there is a gap in the emitted packet stream, it may be necessary to scan more packets in order to fill in the gap. In the latter case, we want to keep the priority queue populated with enough packets to ensure that merging will be successful across traces. If we have completed scanning all traces then we do not require this latter condition to hold.

Because the merging process operates in a progressive fashion, there is some chance that a packet p will be "prematurely" emitted, that is, before all matching copies of p across all traces have been scanned. This will lead to a duplicate of p in the merged trace. In general, it is impossible for us to ensure that we scan all copies of a packet before emitting, unless we read all traces into memory before emitting (effectively setting $\eta = \infty$). In our implementation, we use $\sigma = \eta = 10$ sec in order to bound memory usage. Removal of duplicates is easily corrected by running the merge a second time on the merged trace itself, which will cause any

duplicates to be merged together. In practice we find that our choice of parameters results in very few duplicate packets.

3.2.3 Handling duplicate transmissions

As stated earlier, multiple transmissions of the same packet (say, due to link-layer ARQ) will cause several copies of the same packet to appear in the sniffer traces, and complicates the merging process. Denote $\mathrm{dups}_i(p)$ as the number of duplicates of packet p seen in trace S_i. For example, consider traces S_1 with $\mathrm{dups}_1(p) = 4$ and S_2 with $\mathrm{dups}_2(p) = 2$. The question is how many copies of p should appear in the merged trace $S_1 \cup S_2$.

Without any other information to uniquely identify the multiple copies of p, we opt to adhere to the lower bound, $d^\star = max_i(\mathrm{dups}_i(p))$. We know that *at least* this many copies of p were transmitted, so this is a conservative estimate. Taking the upper bound, $\sum_i \mathrm{dups}_i(p)$, assumes that there is no overlap in the copies of p observed across traces. Our merging process combines all copies of p seen in the input traces and attaches the value of $d^\star$ to the packet in the merged trace.

Two pieces of information could be used to improve this estimate. The TinyOS CC2420 radio stack includes the `tos_dsn` field in each packet, which is incremented for each individual packet transmission. Taking $d' = \max(\texttt{tos\_dsn}_p) - \min(\texttt{tos\_dsn}_p)$ improves upon our lower bound. Unfortunately, the MicaZ implementation of the TinyOS serial stack rewrites packets into an older format that does not include this field, so this information is not available when using a MicaZ as a sniffer.[1] One can also improve the estimate of $d^\star$ by estimating the *overlap* between two sniffers S_1 and S_2 based on the fraction of matching (unique) packets in both traces. We leave exploration of these ideas to future work.

4 Trace Analysis

A wide range of analyses can be performed on the merged LiveNet packet trace. In this section, we describe a range of analysis algorithms for reconstructing a sensor network's behavior. 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 for packets transmitted by node n. Over a representative time interval t, we can compute

$$C_i(n) = \frac{\sum_t \text{packets from } n \text{ received by } S_i}{\sum_t \text{packets transmitted by } n}$$

Likewise, $C(n)$ can be computed for the merged trace $\cup_i S_i$. This calculation requires an estimate of the number of packets actually transmitted by each node n during the time interval. This information can be determined in several ways: for example, using packet-level sequence numbers, or knowledge of the application transmission behavior (e.g., if the application transmits a periodic beacon packet).

This analysis assumes 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

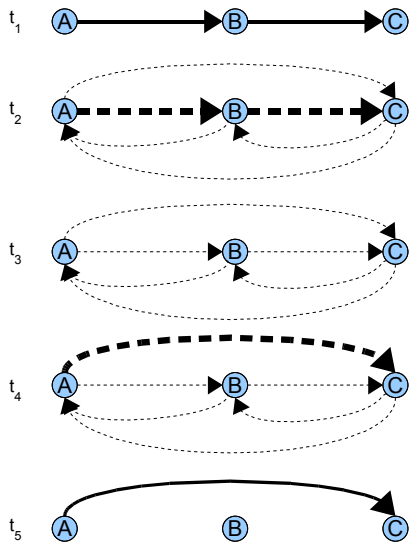
Reconstructing radio connectivity between nodes is seemingly straightforward: for each packet from node a to b, we record an edge $a \rightarrow b$ in the connectivity graph. However, this approach may not reconstruct the *complete* topology, since two nodes a and b within radio range may choose not 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], our goal is only to 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.

4.4 Routing path inference

One of the more interesting analyses involves reconstructing the routing path taken by a packet traveling from a source

[1] Although this could be fixed in the TinyOS code, our sniffer traces from the live disaster drill used the original MicaZ serial stack and therefore are lacking the `tos_dsn` field.



	{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_5, 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.

Figure 2: **Path inference example.**

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. Our goal of our path inference algorithm is to determine the *most probable* routing path $P(s,d,t) = (s, n_1, n_2, ...n_k, d)$ at a given time t based on a possibly incomplete trace.

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. The probability of a link being part of the routing path is computed as a combination of the positive and negative scores, as described below.

Figure 2 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 packet transmission from a to b, we set $\Pi^+(a,b,t) = 1$. For each *conflicting link* $a' \rightarrow b'$, we set $\Pi^-(a',b',t) = -1$. A link conflicts with $a \rightarrow b$ if it shares one endpoint in common (i.e., $a = a'$ or $b = b'$); $b \rightarrow a$ is conflicted by definition as well.

Once the scan is complete, we have a sparse matrix representing the values of Π^+ and Π^- that correspond to observed packet transmissions. However, there are potentially many gaps in the packet trace and we must assign values for the positive and negative scores for times when no transmission on a link is observed. Our approach is to *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 $t_f = t + \delta_f$ and $t_b = t - \delta_b$ that have concrete observations $\Pi^+(a,b,t_f) = \Pi^+(a,b,t_b) = 1$. We then set:

$$\begin{aligned}\Pi^+(a,b,t) \quad = \quad & \max(\max(0, \Pi^+(a,b,t_f) - s \cdot \delta_f), \\ & \max(0, \Pi^+(a,b,t_b) - s \cdot \delta_b))\end{aligned}$$

That is, we take the maximum value of Π^+ time-dilated backwards from t_f or forwards from t_b, capping the value to ≥ 0. Here, s is a scaling constant that determines how quickly the score degrades per unit time. Similarly, we fill in values for missing $\Pi^-(a,b,t)$ values, also capping them to be ≤ 0. In our implementation we set $s = 0.1$.

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 $\Pi(a,b,t)$:

$$\Pi(a,b,t) = \begin{cases} \Pi^+(a,b,t) & \text{if } \Pi^+(a,b,t) \geq |\Pi^-(a,b,t)| \\ \Pi^-(a,b,t) & \text{otherwise} \end{cases}$$

That is, the score is assigned to either the positive or negative score, depending on which has the greater absolute

value. Note that for links for which we have no information, $\Pi(a, b, t) = 0$.

We now have a complete matrix representing the score for each link at each time window. 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:

$$P^{\star}(s, d, t) = \arg \max_{\forall P(s,d,t)} \frac{\sum_{l=\{n1,n2\} \in P(s,d,t)} \Pi(n_1, n_2, t)}{|P(s, d, t)|}$$

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 linear 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 $n_1 \rightarrow n_2$ 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 many 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. However, we cannot guarantee that this algorithm always converges on a unique solution for each time window.

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. 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.

Figure 3: **The indoor treatment area of the disaster drill.** *Faces are blurred to preserve anonymity. Inset shows the electronic triage tag, consisting of an MSP430 processor, CC2420 radio, status LEDs, display, and pulse oximeter for measuring patient vital signs.*

6 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.[2] Disaster response and emergency medicine offer an exciting opportunity for use of wireless sensor networks in a highly dynamic and time-critical environment.

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 a laptop base station located at the incident command post nearby. The 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). 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, as shown in Figure 3. The ECG node consists of a TMote Sky with a custom sensor board providing a two-lead (single-channel) electrocardiograph signal

[2]Anonymity rules for double-blind review prevent us from providing complete details on the deployment and software used.

that is digitized by the mote for transmission over the radio. In addition to the patient sensor nodes, a number of static repeater nodes were deployed to assist with maintaining network connectivity.

The sensor nodes all ran a custom application, implemented in TinyOS, that includes a specialized *ad hoc* routing protocol, a sensor discovery protocol, and a high-level query interface. The routing protocol forms a spanning tree rooted at the base station. Although its design is similar to existing protocols such as *MultihopLQI*, our protocol includes a number of enhancements to improve reliability, including application-layer acknowledgments, a well-tuned route selection algorithm, and careful buffer management. Because sensor nodes in the disaster drill are highly mobile, the protocol adapts quickly to changes in the network topology.

The sensor discovery protocol allows sensor nodes and the base station to discover other nodes in the network and learn of their capabilities. It is based on a simple periodic network flood in which each node transmits its node ID and attached sensor types. The query interface provides a mechanism for injecting queries into the network that specify a node ID, sensor type, sampling rate, expiry time, and optional trigger parameters (for example, whether data should be transmitted only when vital signs fall outside of a given range).

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 7 separate sniffer nodes attached to 5 laptops (two of the laptops had two sniffers to improve yield).

Figure 3 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).

7 Validation Study

The goal of our validation study is the 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 testbed consists of 184 TMote Sky nodes deployed over several floors of an office 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.

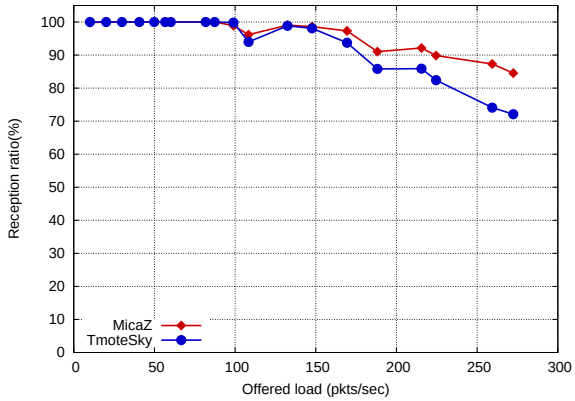


Figure 4: **Sniffer reception rate vs. offered load.** *Data is shown for both TMote Sky and MicaZ sniffer nodes.*

Num traces	Merge time (sec)
2	301
5	616
10	1418
15	1518
20	2143
25	2859

Figure 5: **Merge time as the number of traces varies.** *Each trace represents 1000 sec of packet data of 63 nodes transmitting at 4 Hz each.*

7.1 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 4 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 packet 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.

7.2 Merge performance

Although LiveNet's sniffer traces are intended for offline analysis, the performance of the trace merging process is potentially of interest. Figure 5 shows the performance of the

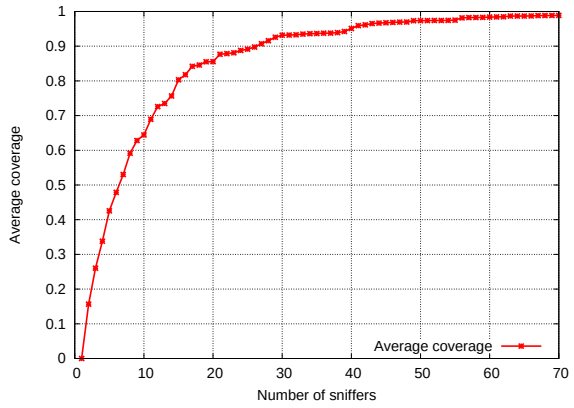


Figure 6: **Sniffer coverage.** *This plot shows the fraction of packets received by the sniffer infrastructure as the number of sniffers is varied. Each point represents an average of 5 random configurations. Coverage of 90% is achieved with 27 sniffers.*

trace merge for an increasing number of sniffer logs. For this experiment, half of the testbed nodes transmit packets at a rate of 4 Hz each, and the other half act as sniffers. Each trace represents 1000 sec of packet data. Merging was performed on an otherwise unloaded 2.4 GHz Linux desktop with 1 GB of memory. Input traces were stored on a remote NFS filesystem accessed via 100 Mbps Ethernet.

As the figure shows, the "break even" point where merge time exceeds the length of the trace is between 5 and 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.

7.3 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 $\binom{N}{70}$ subsets of traces, merge them, and compute the resulting coverage. Instead, we choose five random permutations of the traces and successively merge them, adding one trace at a time to the merge and computing the coverage. We then take the average of the five coverage values for each value of N. The results are shown in Figure 6.

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 $5226\mathrm{m}^2$ (56,252 sq ft). Assuming nodes are uniformly distributed in this area (which is not the case), this suggests that approximately one sniffer per $193\mathrm{m}^2$ (2077 sq ft) would

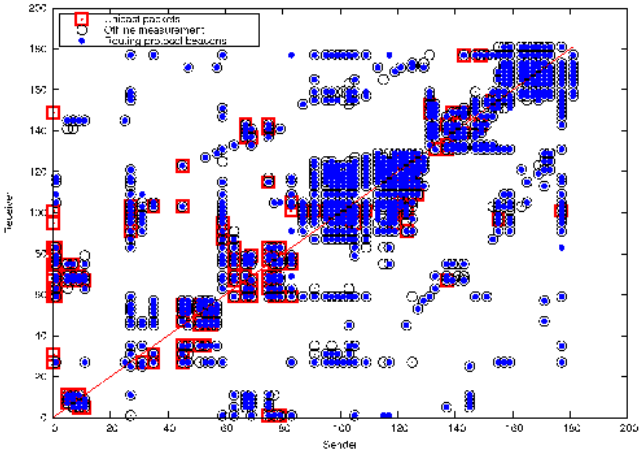


Figure 7: **Network topology reconstruction.** *This figure shows the derived network topology from three sources: (a) unicast transmissions along a spanning tree; (b) extracted neighbor sets from routing protocol beacon messages; and (c) an offline connectivity measurement. Each point represents a single* (sender, receiver) *pair. As the data shows, there is good correspondance between the beacon messages and offline connectivity data, and very few links are discovered by observing unicast transmissions alone.*

	# links	*Offline*	*Beacons*	*Unicasts*
Offline	835	-	777 (93%)	134 (16%)
Beacons	799	777 (97%)	-	134 (17%)
Unicasts	146	134 (92%)	134 (92%)	-

Figure 8: **Comparison of topology data sources.** *Each cell of this table shows the number of* matching *links between each of three information sources: offline connectivity measurement, routing protocol beacons, and unicast packets.*

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 good coverage with fewer nodes.

7.4 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. Note that this can be avoided by reading in each of the input traces into memory prior to merging, however, the memory requirements of this approach are prohibitive. Performing a second "self-merging" pass on the trace can identify and correct duplicate and reordered packets.

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.

7.5 Topology reconstruction

We are interested in determining how well the LiveNet infrastructure can recover the topology of our sensor network testbed. For this experiment, we run the vital sign monitor-

ing application on half of the testbed nodes and sniffers on the other half. 10 nodes were selected as patient sensors and routed data along a spanning tree to a single sink node. There are 65 total nodes running the sensor application.

We use LiveNet to observe two kinds of packets: *unicast transmissions* as packets traverse the spanning tree, and periodic *neighbor table beacons* used by the application's routing protocol. Note that, in general, we expect the neighbor table beacons to reveal many more links than those traversed by unicast packets. We compare the LiveNet-derived topology to data obtained using an offline *connectivity measurement*, similar to SCALE [1], in which each node in turn transmits a series of packets, and all other nodes record the number of packets received from each source. We consider this data to be "ground truth." The connectivity measurement was performed immediately after we completed the LiveNet run.

Figure 7 shows the complete set of links determined using LiveNet and the offline connectivity measurement. There are several interesting features of this graph. First, the unicast packets reveal only a small number of links (146 in this case). Second, network connectivity exhibits a fair degree of asymmetry. Third, while there are a number of links not observed by LiveNet, there are also links *only* seen in LiveNet and not in the connectivity measurement. Figure 8 gives a breakdown of the number of links from each source of information. 22 links are seen only in the routing beacon messages and not in the "ground truth" data set. We suspect this is due to poor link quality and the limited number of per-node transmissions (10) used by the connectivity measurement.

7.6 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 taken in advance, we can compare the output of the path inference algorithm to ground truth.

Figure 9 shows the results. The top portion of the figure shows the routing path determined by the path inference algorithm. Large points indicate routes that were fully observed by the LiveNet infrastructure, while small points indicate routes containing one or more inferred hops. The lower portion of the figure shows the corresponding Π score for the chosen path.

The paths inferred by our algorithm correspond to ground truth in all but a few cases. For example, at times $t = 300$, 600, 1200, 1500, and 1800, there is an inferred path (with a relatively low Π score) between two observed paths. This is because LiveNet observes the path switch as it is taking place, so we briefly infer the existence of an "intermediate" route. As expected the path switches occur on 5 minute intervals.

8 Deployment evaluation

Finally, we perform an evaluation of the LiveNet traces gathered during the disaster drill described in Section 6.

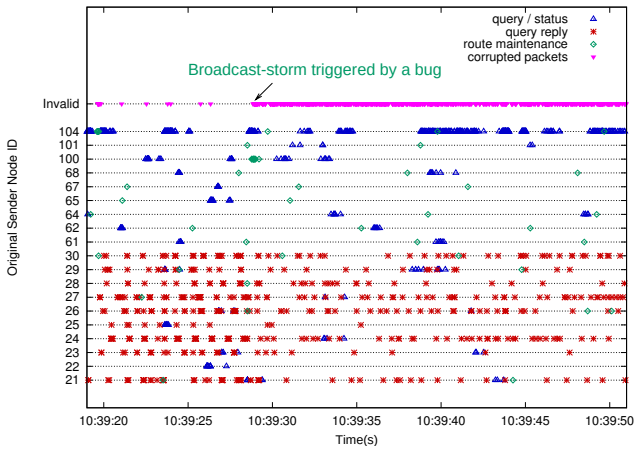


Figure 11: **Packet type breakdown for a portion of the disaster drill.** *The graph shows the sender and type for each packet during a portion of the disaster drill. Periodic status and route maintenance messages can be seen along with query replies. Packets with invalid header fields are also shown; the broadcast storm can be seen starting at 10:39:29.*

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 10 and 11. 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 10 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 11, 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 10.

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. A node could conceivably fail to transmit a status message due to excessive CSMA backoff, so the results here are somewhat conservative.

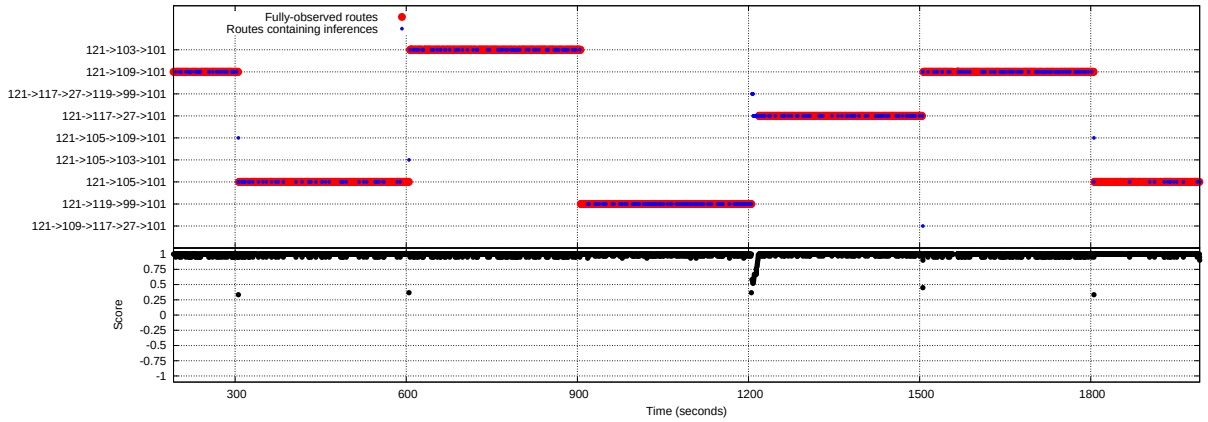


Figure 9: **Path inference validation in the testbed.** *This figure shows a plot of the inferred routing path for an experiment in which a node is programmed to use a different routing path every 5 minutes. The top portion of the figure indicates the inferred path, and the lower portion of the figure shows the mean* Π *score over the inferred path (*$\Pi = 1$ *indicates a complete observation of this path at the given time).*

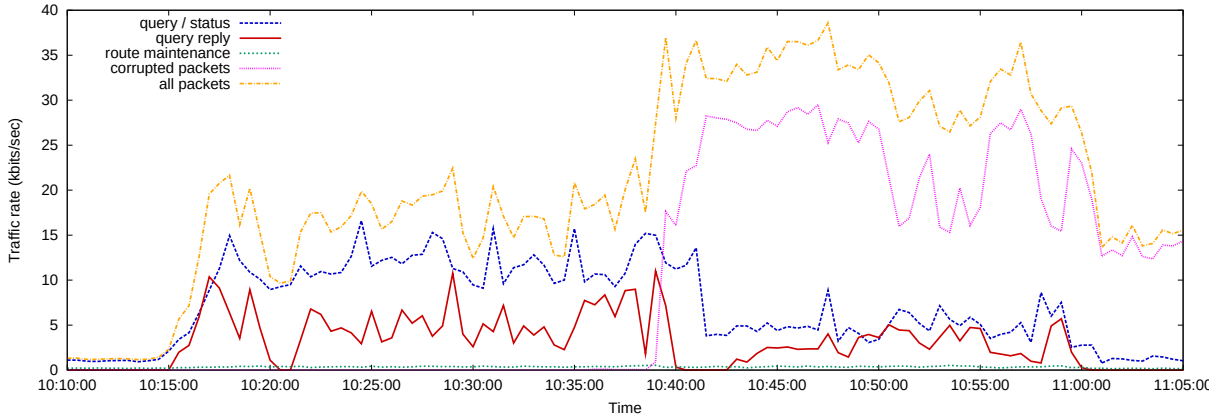


Figure 10: **Overall traffic rate during the disaster drill.** *The total traffic is averaged over 30 sec windows and is broken down by type: queries/status packets, query reply packets, route maintenance packets, and corrupted packets. The broadcast storm begins at time 10:39.*

Figure 12 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 laying 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. Visualizing such a complex dataset presents challenges of its own, but to give an example we show the partial topology consisting only of observed unicast links in Figure 13. The full connectivity graph including all neighbor relationships is too dense to show here. For each link, we evaluate the ETX metric [4], defined as the mean number of transmissions of a given packet along each link; our routing protocol uses ARQ with a maximum retransmission count of 5. This data can be used to estimate radio link quality.

A few observations can be made from this figure. First, most nodes use several outbound links, indicating a fair amount of route adaptation. Keep in mind that 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 they were used for relaying packets. We also see one of the sink nodes (node 100) relaying packets as well.

In order to determine "hotspots" in the network, we look at 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 14 shows the breakdown by node ID and packet type. Since we are counting all transmissions by a node, these totals include forwarded packets; this explains the wide variation in traffic.

The graph reveals a few interesting features. the repeater nodes generated a larger amount of traffic overall than the

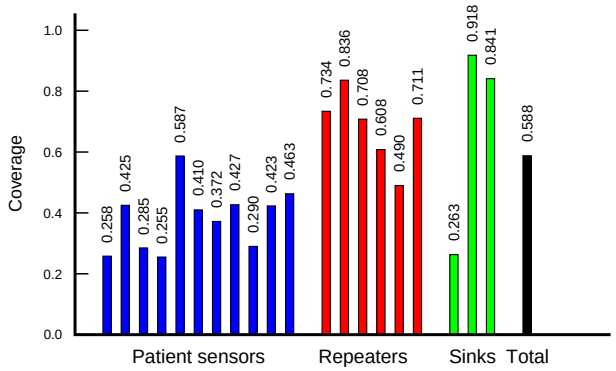


Figure 12: **Per-node coverage during the drill.** *This graph shows the coverage of the LiveNet sniffers for each of the 20 nodes in the disaster drill. Coverage for mobile patient sensors is somewhat lower than for the fixed repeater nodes. The overall coverage is 58%.*

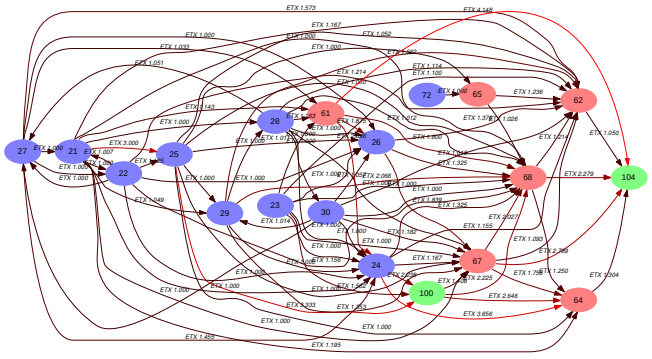


Figure 13: **Unicast links observed during the disaster drill.** *Data is shown only for the first half of the drill, before all nodes were moved indoors together. Each edge is labelled with the mean number of packet transmissions (ETX) along the link. Blue nodes are patient sensors, red nodes are repeaters, and green nodes are sinks.*

patient sensors; this indicates that the repeaters were heavily used, as suggested by the topology shown in Figure 13. 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 4088 packets; instead we infer a total of 15017. Node 62 may have then dropped as many as $(15017 - 4088)/15017 = 72\%$ of the query replies it was forwarding. With no further information, the totals in Figure 14 therefore represent a conservative upper bound on the total traffic transmitted by the node.

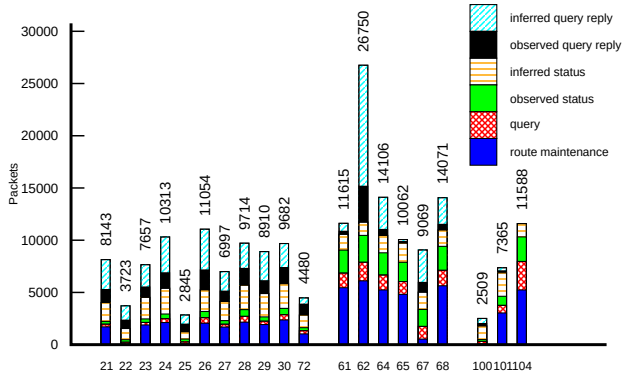


Figure 14: **Inferred per-node packet traffic load during the drill.** *This graph shows the number of packets transmitted by each node during the drill. Both observed traffic and inferred traffic based on sequence number analysis are shown in the graph.*

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 crisp as those in Figure 9. We take as an example 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 15. 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, as well as several other patient sensors. The longest routing path inferred 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 on the disaster drill traces involves understanding the causes of data loss from sources to sinks. We found that the overall data yield during the drill was very low, averaging about 20% across the patient sensors. There are several possible sources of data loss in our system: packet loss along routing paths, node failures or reboots, or query timeout.

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. Our system uses a lease model 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 sec before the lease expires; however, if this query is not received by the node in time, the query will

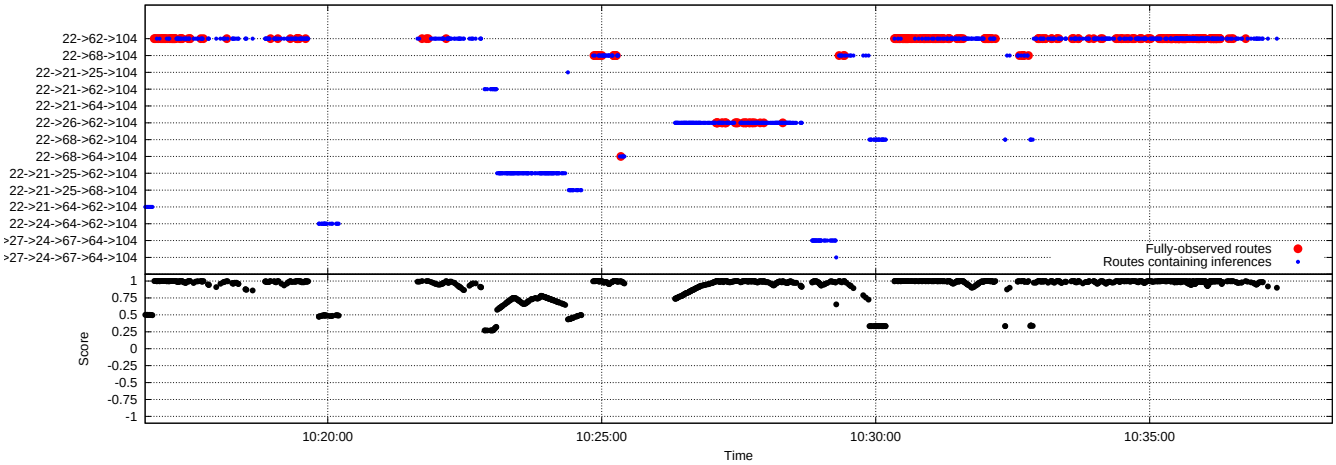


Figure 15: **Routing paths for one of the patient sensors during the disaster drill.**

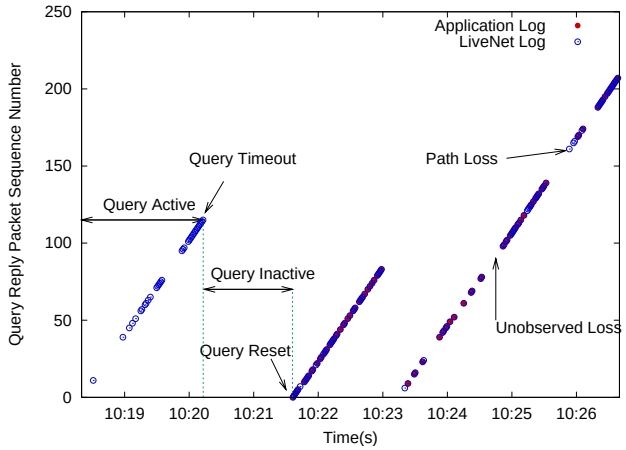


Figure 16: **Query behavior.** *This figure shows the behavior of an individual query during the drill. The plot labels query reply packets received at the sink versus packets received by the sniffers, indicating packet loss along the routing path. Sequence number resets can be caused by a query timeout or a node reboot, also labelled in the figure.*

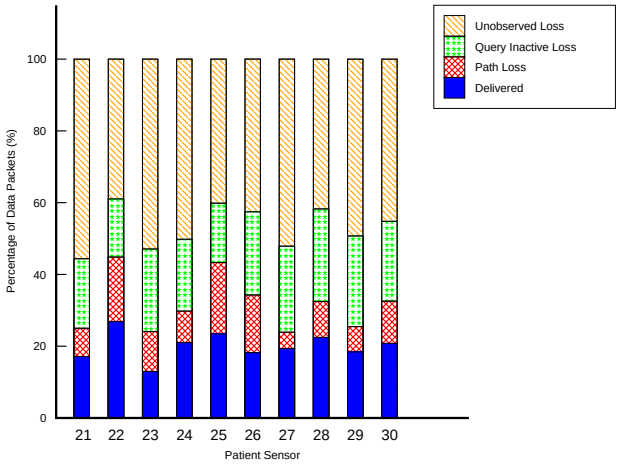


Figure 17: **Query yield per node during the disaster drill.** *The overall data yield during the drill was quite low. Using the LiveNet infrastructure, we were able to break down the causes of data loss in terms of packet loss and query timeout.*

time out until the sink re-issues the query.

Figure 16 shows the behavior of an individual node during the drill, combining LiveNet traces with an application-level packet trace collected at the sink node. From this graph we can infer time periods when queries were active or inactive, query timeouts, and path loss to the sink (that is, a packet observed by LiveNet but not received at 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.

Using this combined information, we can then break down the data loss in terms of path loss, inactive query periods (caused by query timeouts), and *unobserved loss*; that is, packets that were neither observed by LiveNet or the sink. Figure 17 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. However, it is also possible that nodes failed to transmit these packets at all, due to excessive CSMA backoff.

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 is a first step towards 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.

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, 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.

[3] 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.

[4] D. S. J. De Couto, D. Aguayo, J. Bicket, and R. Morris. A high-throughput path metric for multi-hop wireless routing. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom '03)*, San Diego, California, September 2003.

[5] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proc. Second ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[6] A. Jardosh, K. Ramachandran, K. C. Almeroth, , and E. M. Belding-Royer. Understanding congestion in ieee 802.11b wireless networks. In *Proc. the Internet Measurement Conference(IMC 2005)*, 2005.

[7] 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.

[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] 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.

[11] M. Ringwald, M. Cortesi, K. Römer, and A. Vialetti. Demo abstract: Passive inspection of deployed sensor networks with snif. In K. Langendoen and T. Voigt, editors, *Adjunct Proceedings of the 4th European Confererence on Wireless Sensor Networks (EWSN 2007)*, pages 45–46, Delft, The Netherlands, Jan. 2007.

[12] M. Rodrig, C. Reis, R. Mahajan, D. Wetherall, and J. Zahorjan. Measurement-based characterization of 802.11 in a hotspot setting. In *E-WIND '05: Proceeding of the 2005 ACM SIGCOMM workshop on Experimental approaches to wireless network design and analysis*, pages 5–10, New York, NY, USA, 2005. ACM Press.

[13] S. Rost and H. Balakrishnan. Memento: A Health Monitoring System for Wireless Sensor Networks. In *IEEE SECON*, Reston, VA, September 2006.

[14] V. Shnayder, M. Hempstead, B. rong Chen, G. Werner-Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, November 2004.

[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.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. the 5th OSDI*, 2002.

[19] J. Yeo, M. Youssef, and A. Agrawala. A framework for wireless lan monitoring and its applications. In *Proc. ACM Workshop on Wireless Security (WiSe 2004)*, 2004.