IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

LiMoSense —Live Monitoring in Dynamic Sensor Networks

Ittay Eyal, Idit Keidar and Raphael Rom

CCIT Report #786
March 2011

Electronics
Computers
Communications

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #786 March 2011

LiMoSense – Live Monitoring in Dynamic Sensor Networks

Ittay Eyal Idit Keidar Raphael Rom
Department of Electrical Engineering,
Technion — Israel Institute of Technology

Abstract

We present LiMoSense, a fault-tolerant live monitoring algorithm for dynamic sensor networks. LiMoSense uses gossip to dynamically track and aggregate a large collection of ever-changing sensor reads. It overcomes message loss, node failures and recoveries, and dynamic network topology changes. We formally prove correctness of LiMoSense; we use simulations to illustrate its ability to quickly react to network and value changes and provide accurate information.

1 Introduction

The subject of environmental monitoring is gaining increasing interest in recent years. Monitoring is necessary for research, and it is critical for protecting the environment by quickly discovering fire outbreaks in distant areas, cutting off electricity in the event of an earthquake, etc. In order to perform these tasks, it is necessary to perform constant measurements in wide areas, and collect this data quickly. In years to come, we can expect to see sensor networks with thousands of light-weight nodes monitoring conditions like seismic activity, humidity or temperature [1, 2]. Each of these nodes is comprised of a sensor, a wireless communication module to connect with close-by nodes, a processing unit and some storage. The large number of nodes prohibits a centralized solution in which the raw monitored data is accumulated at a single location. Specifically, all sensors cannot directly communicate with a central unit.

Fortunately, often the raw data is not necessary. Rather, an *aggregate* that can be computed *inside the network*, such as the sum or average of sensor reads, is of interest. For example, when measuring rainfall, one is interested only in the total amount of rain, and not in the individual reads at each of the sensors. Similarly, one may be interested in the average humidity or temperature rather than minor local irregularities.

In such settings, it is particularly important to perform *live monitoring*, that is, to constantly obtain a timely and accurate picture of the ever-changing data. For example, for disaster avoidance one must quickly propagate changes such as a rapid temperature incline that indicates a fire outbreak, or an increase in seismic activity that indicates an earthquake. In this paper we introduce and tackle this problem of live monitoring in a dynamic sensor networks. This problem is particularly challenging due to the dynamic nature of sensor networks, where nodes may fail and may be added on the fly, and the network topology may change due to battery decay or weather change. The formal model and problem definition appear in Section 3.

Although many works have dealt with data aggregation in sensor networks, they have concentrated on a single aggregation iteration, assuming the sensor reads do not change during this

iteration [3, 4, 5, 6] (for further details on previous work, see Section 2). Though it is in principle possible to perform live monitoring using multiple iterations of such algorithms, this approach is not adequate, due to the inherent tradeoff it induces between accuracy and speed of detection. If one periodically restarts the aggregation, then it may take a long time to learn of changes that occur between restarts. If, on the other hand, there are no changes between restarts, then resources are wasted running the redundant aggregation. Alternatively, if one starts a new iteration every time a data change is discovered at one of the nodes, then in a rapidly changing network, the aggregation algorithm might repeatedly restart without ever getting a chance to complete its operation.

In Section 4 we present our new **Li**ve **Mo**nitoring for **Sens**or networks algorithm, LiMoSense. Our algorithm computes the average over a dynamically changing collection of sensor reads, i.e., it solves the *live average monitoring* problem. At its core, LiMoSense employs gossip-based aggregation [3, 5], with a new approach to accommodate data changes while the aggregation is on-going. This is tricky, because when a sensor read changes, its old value should be removed from the system after it has propagated to other nodes. LiMoSense further employs a new technique to accommodate message loss, failures, and dynamic network behavior in asynchronous settings. This is tricky again, since a node cannot know whether a previous message it had sent over a faulty or lossy link has arrived or not.

In Section 5, we prove that once the network stabilizes, in the sense that no more value or topology changes occur, LiMoSense eventually converges to the correct average, despite message loss. Obviously, we cannot bound the algorithm's convergence time in a completely asynchronous system. Instead, we analyze convergence time in a *synchronous uniform* run, where all nodes take steps at the same average frequency. We show that in such runs, once the system stabilizes, the estimates nodes have of the desired value converge exponentially fast (i.e., in logarithmic time).

To demonstrate the effectiveness of LiMoSense in various dynamic scenarios, we present in Section 6 results of extensive simulations, showing its quick reaction to dynamic data read changes and fault tolerance. We compare our algorithm to a periodically restarting aggregation algorithm.

In summary, this paper makes the following contributions:

- It presents LiMoSense, the first live monitoring algorithms for highly dynamic and error-prone environments (Section 4).
- It proves correctness of the algorithm, namely robustness and eventual convergence (Section 5).
- It shows, through analysis and simulation, that LiMoSense converges exponentially fast (Section 5).
- It demonstrates by extensive simulation the algorithm's efficiency and fault-tolerance in various dynamic scenarios (Section 6).

2 Related Work

To gather information in a sensor network, one typically relies on in-network *aggregation* of sensor reads. The vast majority of the literature on aggregation has focused on obtaining a *single* summary of sensed data, assuming these reads do not change while the aggregation protocol is running [6, 3, 4, 5]. The only exception we are aware of is work on dynamic aggregation by Birk et al. [7]; however,

this solution is limited to unrealistic settings, namely a static topology with reliable communication links, failure freedom, and synchronous operation.

For obtaining a single aggregate, two main approaches were employed. The first is hierarchical gathering to a single base station. This approach is used in TAG [6]. The hierarchical method incurs considerable resource waste for tree maintenance in the face of topology changes. Moreover, the use of tree aggregation in dynamic environments results in aggregation errors, as shown in [8].

The second approach is gossip-based aggregation at all nodes. To avoid counting the same data multiple times, Nath et al. [9] employ order and duplicate insensitive (ODI) functions to aggregate inputs in the face of message loss and a dynamic topology. However, these functions do not support dynamic inputs or node failures. Moreover, due to the nature of the ODI functions used, the algorithms' accuracy is inherently limited – they do not converge to an accurate value [10].

An alternative approach to gossip-based aggregation is presented by Kempe et al. [3]. They introduce Push-Sum, an average aggregation algorithm, and show that it converges exponentially fast in fully connected networks where nodes operate in lock-step. Shah et al. analyze this algorithm in an arbitrary topology [4]. Jesus et al. [11] employ a similar approach enhanced with flow tracking to overcome message loss. Although these algorithms do not deal with dynamic inputs and topology as we do, we borrow some techniques from them. In particular, our algorithm is inspired by the Push-Sum construct, and operates in a similar manner in static settings. We analyze its convergence speed when the nodes operate independently.

It was noted in [5] that any function that can be separated to calculating sums and averages of the input values can be efficiently calculated with an average aggregation algorithm. LiMoSense can therefore be used to perform live monitoring of such separable functions.

Note that aggregation in sensor networks is distinct from other aggregation problems, such as stream aggregation, where the data in a sliding window is summarized. In the latter, a single system component has the entire data, and the distributed aspects do not exist.

3 Model and Problem Definition

3.1 Model

The system is comprised of a dynamic set of nodes (sensors), partially connected by dynamic undirected communication links. Two nodes i and j connected by a link are called *neighbors*, and they can send messages to each other. These messages either arrive at some later time, or are lost. Messages that are not lost on each link arrive in FIFO order. Links do not generate or duplicate messages.

The system is asynchronous and progresses in steps, where in each step an event happens and the appropriate node is notified, or a node acts spontaneously. When performing actions due to notifications or spontaneously, nodes may change their internal state and send messages to their neighbors.

Nodes can be dynamically added to the system, and may fail or be removed from the system. The set of nodes at time t is denoted $\mathcal{N}_t$. The *system state* at time t consists of the internal states of all nodes in $\mathcal{N}_t$, and the links among them. When a node is added (`init` event), it is notified, and its internal state becomes a part of the system state. When it is removed (`remove` event), it is not allowed to perform any action, and its internal state is removed from the system state.

Each sensor has a time varying *data read* in $\mathbb{R}$. A node's initial data read is provided as a

parameter when it is notified of its `init` event. This value may later change (`change` event) and the node is notified with the newly read value. For a node i in $\mathcal{N}_i$, we denote[1] by r_i^t, the latest data read provided by an `init` or `change` event at that node before time t.

Communication links may be added or removed from the system. A node is notified of link addition (`addNeighbor` event) and removal (`removeNeighbor` event), given the identity of the link that was added/removed. We call these *topology events*. For convenience of presentation, we assume that initially, nodes have no links, and they are notified of their neighbors by a series of `addNeighbor` events. We say that a link (i, j) is *up* at step t if by step t, both nodes i and j had received an appropriate `addNeighbor` notification and no later `removeNeighbor` notification. Note that a link (i, j) may be "half up" in the sense that the node i had been notified of its addition but node j was not, or if node j had failed.

A node may send messages on a link only if the last message it had received regarding the state of the link is `addNeighbor`. If this is the case, the node may also receive a message on the link (`receive` event). The node is notified of the event with the received message.

Global Stabilization Time

In every run, there exists a time called *global stabilization time*, GST, from which onward the following properties hold:

1. The system is *static*, i.e., there are no `change`, `init`, `remove`, `addNeighbor` or `removeNeighbor` events.
2. If the latest topology event a node $i \in \mathcal{N}_{\text{GST}}$ has received for another node j is `addNeighbor`, then node j is alive, and the latest topology event j has received for i is also `addNeighbor` (i.e. there are no "half up" links).
3. The network is connected.
4. If a link is up after GST, and infinitely many messages are sent on it, then infinitely many of them arrive.

3.2 The Live Average Monitoring Problem

We define the *read average* of the system at time t as

$$R^t \triangleq \frac{1}{|\mathcal{N}_t|} \sum_{i \in \mathcal{N}_t} r_i^t \quad .$$

Note that the read average does not change after GST. Our goal is to have all nodes estimate the read average after GST. More formally, an algorithm solving the *Live Average Monitoring Problem* gets time-varying data reads as its inputs, and has nodes continuously output their *estimates* of the average, such that at every node in $\mathcal{N}_{\text{GST}}$, the output estimate converges to the read average after GST.

[1]For any variable, the node it belongs to is written in subscript and, when relevant, the time is written in superscript.

Metrics

We evaluate live average monitoring algorithms using the following metrics: (1) *Mean square error*, *MSE*, which is the mean of the squares of the distances between the node estimates and the read average; and (2) ε-*inaccuracy*, which is the percentage of nodes whose estimate is off by more than ε.

4 The LiMoSense Algorithm

We describe the operation of our algorithm for performing live average monitoring in a sensor network. The algorithm is based on Push-Sum, and introduces mechanisms for live updates following changes of sensor reads, and for overcoming message loss, link failures and node failures.

In Section 4.1 we describe a simplified version of the algorithm for dynamic inputs but static topology and no failures. Then, in Section 4.2, we describe the complete robust algorithm.

4.1 Failure-Free Algorithm

4.1.1 Overview

We begin by describing a version of the algorithm that handles dynamically changing inputs, but assumes no message loss, and no link or node failures. This algorithm is shown in Algorithm 1.

The base of the algorithm operates like Push-Sum. Each node maintains a weighted estimate of the read average (a pair containing the estimate and a weight), which is updated as a result of the node's communication with its neighbors. As the algorithm progresses, the estimate converges to the read average. In the absence of value changes, this algorithm behaves like Push-Sum, which was indeed shown to converge to the correct read average in static networks [3, 4].

A node whose read value changes must notify the other nodes. It need not only introduce the new value, but also needs to undo the effect of its previous read value, which by now has partially propagated through the network.

4.1.2 Details

The algorithm often requires nodes to merge two weighted values into one. They do so using the *weighted value sum* operation, which we define below and concisely denote by $\oplus$:

$$\langle v_a, w_a \rangle \oplus \langle v_b, w_b \rangle \triangleq \langle \frac{v_a w_a + v_b w_b}{w_a + w_b}, w_a + w_b \rangle \ . \tag{1}$$

Note that weighted value sum is a commutative and associative operation. Subtraction operations will be used later, they are denoted by $\ominus$ and are defined as:

$$\langle v_a, w_a \rangle \ominus \langle v_b, w_b \rangle \triangleq \langle v_a, w_a \rangle \oplus \langle v_b, -w_b \rangle \ . \tag{2}$$

The state of a node (lines 2–3) consists of a weighted value, $\langle est_i, w_i \rangle$, where est_i is an output variable holding the node's estimate of the read average, and the value $prevRead_i$ of the latest data read. We assume at this stage that each node knows its static set of neighbors. We shall remove this assumption later, in the robust LiMoSense algorithm.

Algorithm 1: Failure Free

```
1  state
2      ⟨est_i, w_i⟩ ∈ ℝ
3      prevRead_i ∈ ℝ
4  on init_i(initVal)
5      ⟨est_i, w_i⟩ ← ⟨initVal, 1⟩
6      prevRead_i ← initVal
7  periodically send_i()
8      Choose a neighbor j uniformly at random.
9      w_i ← w_i/2
10     send (⟨est_i, w_i⟩) to j
11 on receive_i(⟨v_in, w_in⟩) from j
12     ⟨est_i, w_i⟩ ← ⟨est_i, w_i⟩ ⊕ ⟨v_in, w_in⟩
13 on change_i(newRead)
14     est_i ← est_i + (1/w_i) · (newRead − prevRead_i)
15     prevRead_i ← newRead
```

Node i initializes its state on its `init` event. The data read is initialized to the given value *initVal*, and the estimate is $\langle initVal, 1\rangle$ (lines 5–6).

The algorithm is implemented with the functions `receive` and `change`, which are called in response to events, and the function `send`, which is called periodically.

Periodically, a node i shares its estimate with a neighbor j chosen uniformly at random (line 8). It transfers half of its estimate to node j by halving the weight w_i of its locally stored estimate and sending the same weighted value to that neighbor (lines 9-10). When the neighbor receives the message, it merges the accepted weighted value with its own (line 12).

Correctness of the algorithm in static settings follows from two key observations. First, *safety* of the algorithm is preserved, because the system-wide weighted average over all weighted-value estimate pairs at all nodes and all communication links is always the correct read average; this invariant is preserved by send and receive operations. Thus, no information is "lost". Second, the algorithm's *convergence* follows from the fact that when a nodes merges its estimate with that received from a neighbor, the result is closer to the read average.

We proceed to discuss the dynamic operation of the algorithm. When a node's data read changes, the read average changes, and so the estimate should change as well. Let us denote the previous read of node i by r_i^{t-1} and the new read at step t by r_i^t. In essence, the new read, r_i^t, should be added to the system-wide estimate with weight 1, while the old read, r_i^{t-1}, ought to be deducted from it, also with weight 1. But since the old value has been distributed to an unknown set of nodes, we cannot simply "recall" it. Instead, we leverage the algorithm's convergence property, and make the appropriate adjustment locally, allowing the natural flow of the algorithm to propagate it.

We now explain how we compute the local adjustment. The system-wide estimate should move by the difference between the read values, factored by the relative influence of a single sensor, i.e., $1/n$. To achieve this, we could shift a weight of 1 by $r_i^t - r_i^{t-1}$. Alternatively, we can shift a weight

of w by this difference factored by $1/w$. Therefore, in response to a `change` event at time t, if the node's estimate before the change was est_i^{t-1} and its weight was w_i^{t-1}, then the estimate is updated to (lines 14-15)

$$est_i^t = est_i^{t-1} + (r_i^t - r_i^{t-1})/w_i^{t-1} \ .$$

4.2 Adding Robustness

4.2.1 Overview

Overcoming failures is challenging in an asynchronous system, where a node cannot determine whether a message it has sent was successfully received.

In order to overcome message loss and link and node failure, each node maintains a summary of its conversations with its neighbors. Nodes interact by sending and receiving these summaries, rather than the weighted values they have sent in the failure-free algorithm. The data in each message subsumes all previous value exchanges on the same link. Thus, if a message is lost, the lost data is recovered once an ensuing message arrives. When a link fails, the nodes at both of its ends use the summaries to retroactively cancel the effect of all the messages transferred over it. A node failure is treated as the failure of all its adjacent links.

Implementing the summary approach naïvely would cause summary sizes to increase unboundedly as the algorithm progresses. To avoid that, we devised a hybrid approach of push and pull gossip that negates this effect without resorting to synchronization assumptions.

4.2.2 Details

The full LiMoSense algorithm, shown as Algorithm 39, is based on the failure-free algorithm. In addition to the state information of the failure-free algorithm, is also maintains the list of its neighbors, and a summary of the data it has sent and received on each of them (lines 5-6). On initialization, a node has no neighbors (lines 10–12).

The `change` function is identical to the one of the failure-free algorithm. The functions `receive` and `send`, however, instead of transferring the weighted values as in the failure-free case, transfer the summaries maintained for the links. In addition, when a node i wishes to send a weighted value to a node j, it may do so using either *push* or *pull*.

When pushing, node i adds the new weighted value to $sent_i(j)$ and sends $sent_i(j)$ to j (lines 14–16). When receiving this summary, node j calculates the received weighted value by subtracting the appropriate *received* variable from the newly received summary (line 27). After acting on the received message (line 28), node j replaces its *received* variable with the new weighted value (line 29). Thus, if a message is lost, the next received message compensates for the loss and brings the receiving neighbor to the same state it would have reached had it received the lost messages as well. Whenever the last message on a link (i, j) is correctly received and there are no messages in transit, the value of $sent_i^j$ is identical the value of $received_j^i$.

Since the weights are (usually) positive, push operations, if used by themselves, cause the *sent* and *received* variables to grow to infinity. In order to overcome that, LiMoSense uses a hybrid push/pull approach, which keeps these weights small without requiring bilateral coordination. A node uses pull operations to decrease the *sent* variables of its neighbors, and thereby its own *received*. The pull message is a request from a neighbor to push back an inverse weighted value.

Algorithm 2: LiMoSense

```
1  state
2      ⟨est_i, w_i⟩ ∈ ℝ
3      prevRead_i ∈ ℝ
4      neighbors_i ⊂ ℕ
5      sent_i : n → (ℝ² × ℝ²) ∪ ⊥
6      received_i : n → (ℝ² × ℝ²) ∪ ⊥
7  on init_i(initVal)
8      ⟨est_i, w_i⟩ ← ⟨initVal, 1⟩
9      prevRead_i ← initVal
10     neighbors_i ← ∅
11     ∀j : sent_i(j) = ⊥
12     ∀j : received_i(j) = ⊥
13 function pushSend_i(sendVal)
14     ⟨est_i, w_i⟩ ← ⟨est_i, w_i⟩ ⊖ sendVal
15     sent_i(j) ← sent_i(j) ⊕ sendVal
16     send (sent_i(j), PUSH), to j
17 periodically send_i()
18     if w_i < 2q then return        (weight min.)
19     Choose a neighbor j uniformly at
       random.
20     type ←
       choose at random from {PUSH, PULL}
21     if type = PUSH then
22         pushSend(⟨est_i, w_i/2⟩)
23     else                           (type = PULL)
24         send (⟨est_i, w_i/2⟩, PULL) to j
25 on receive_i(⟨v_in, w_in⟩, type) from j
26     if type = PUSH then
27         diff ← ⟨v_in, w_in⟩ ⊖ received_i(j)
28         ⟨est_i, w_i⟩ ← ⟨est_i, w_i⟩ ⊕ diff
29         received_i(j) ← ⟨v_in, w_in⟩
30     else                           (type = PULL)
31         pushSend(⟨v_in, −w_in⟩)
32 on change_i(r_new)
33     est_i ← est_i + (1/w_i) · (r_new − prevRead_i)
34     prevRead_i ← r_new
35 on addNeighbor_i(j)
36     neighbors_i ← neighbors_i ∪ {j}
37     sent_i(j) ← ⟨0, 0⟩
38     received_i(j) ← ⟨0, 0⟩
39 on removeNeighbor_i(j)
40     ⟨est_i, w_i⟩ ←
       ⟨est_i, w_i⟩ ⊕ sent_i(j) ⊖ received_i(j)
41     neighbors_i ← neighbors_i \ {j}
42     sent_i(j) ← ⊥
43     received_i(j) ← ⊥
```

In line 20, the algorithm randomly decides whether to perform push or pull[2]. When pulling, i sends the weighted value to j with the PULL flag. Once node j receives the message, it merges it with its own value, and relays i the same weighted pair using the standard push mechanism, but with a *negative* weight (line 31). Thus, the weights of the *sent* and *received* records fluctuate around 0 rather than growing to infinity. Additionally, to prevent infinitesimal weights, a node does not perform a `send` step if the result would bring its weight to be smaller than a quantization constant q.

Upon notification of topology events, nodes act as follows. When notified of an `addNeighbor` event, a node initializes its transfer records *sent* and *received* for this link, noting that 0 weight was transferred in both directions. It also adds the new neighbor to its *neighbors* list (lines 36-38).

When a link notified of a `removeNeighbor` event, a node reacts by nullifying the effect of this link. Pull messages that were sent and/or received on this link had no effect. Nodes therefore need to undo only the effects of sent and received push messages, which are summarized in the respective

[2]We use random choice for ease of presentation. One may choose to perform pull less frequently to conserve bandwidth.

sent and *received* variables. When a node i discovers that link (i, j) has failed, it adds the outgoing link summary $sent_i^j$ to its estimate, thus cancelling the effect of ever having sent anything on the link, and subtracts the incoming link summary $received_i^j$ from its estimate, thereby cancelling the effect of everything it has received (line 40). The node also removes the neighbor from its *neighbors* list and discards its link records (lines 41–43).

After a node joins the system or leaves it, its neighbors are notified of the appropriate topology events, adding links to the new node, or removing links to the failed one. Thus, when a node fails, any parts of its read value that had propagated through the system are annulled, and it no longer contributes to the system-wide estimate.

5 Analysis

In this section, we show that the LiMoSense algorithm adapts to network topology and value changes and converges to the correct average. We start in Section 5.1 by proving safety: we define an invariant of the system state and prove that it holds (even during periods with changes). Then, in Section 5.2, we prove liveness, namely that after GST the estimates at all nodes converge to the average of the latest read values. Liveness is proven for all runs, including asynchronous ones. Clearly, in asynchronous settings, any algorithm's convergence time is inherently unbounded. In Section 5.3, we consider some limitations on scheduling (synchrony), and analyze the convergence time after system stabilization under these realistic though more limited circumstances.

5.1 Safety

We show the existence of an invariant that holds throughout the system's operation. The *estimate average* at time t, E^t, is the weighted average over all nodes of their weighted values, their outgoing link summaries in their *sent* variables and the inverse of their incoming logs in their *received* variables. We denote the *read average* at time t, by R^t. We define the *estimate sum* $\langle E^t, n\rangle$ and the *read sum* $\langle R^t, n\rangle$.

The invariant $\mathcal{I}$ states that the estimate sum equals the read sum:

$$\mathcal{I} \ : \ \langle R^t, n\rangle \ = \ \bigoplus_{i=1}^{n} \langle r_i^t, 1\rangle \ = \ \bigoplus_{i=1}^{n} \left(\langle est_i^t, w_i^t\rangle \oplus \bigoplus_{j \in neighbors_i^t} \left(sent_i^t(j) \ominus received_i^t(j) \right) \right) \ = \ \langle E^t, n\rangle \quad (3)$$

Static Behavior

First, we consider the case where the system is completely static (no topology or value changes), so the only possible events are `send` and `receive`. Note that message loss is not an event and it does not effect the state of the system. Specifically, it does not effect the correctness of this lemma or the following ones.

Lemma 1 (Static operations)**.** *If the invariant $\mathcal{I}$ holds at time $t-1$, and step t is either* `send` *or* `receive`*, then the invariant holds at time t.*

Proof. If step t is a push `send` at node i, then the weighted value $\langle est_i^{t-1}, \frac{1}{2} w_i^{t-1}\rangle$ is subtracted from the weighted value of node i (line 14), and the weighted value $\langle est_i^{t-1}, \frac{1}{2} w_i^{t-1}\rangle$ is added to $sent_i^j$ (line 15). If step t is a push `receive` at node j, then the weighted value $\langle est_{\text{in}}^{t-1}, w_{\text{in}}^{t-1}\rangle$ is added to

the weighted value of node j (line 28), and the same weighted value is subtracted from $received_j^i$ (line 29). If step t is a pull `send` at node i, then the estimate sum remains unchanged (line 24). If step t is a pull `receive`, it is similar to a push `send` (line 31). In all cases, the estimate sum according to Equation 3 is unchanged.

None of these events implies a change to the read sum, therefore, given that the invariant holds at $t-1$, and since neither the read sum nor the estimate sum changed, it also holds at t. □

Dynamic Values

When the value read by node i changes, the node updates its estimate in a manner that changes the estimate sum by the correct distance.

Lemma 2 (Read value change)**.** *If the invariant $\mathcal{I}$ holds at time $t-1$, and step t is* `change`*, then the invariant holds at time t.*

Proof. Recall we denote the previous read of node i by r_i^{t-1} and its new read at step t by r_i^t. After the change of the read value, the new read average is $A_t = \mathrm{A}_{t-1} - \frac{1}{n}r_i^{t-1} + \frac{1}{n}r_i^t$, and the weighted value $\left\langle est_i^{t-1} + \frac{r_i^t - r_i^{t-1}}{w_i}, w_i \right\rangle$ replaces the weighted value of node i (line 33). Therefore, we see that the new estimate sum is equal to the new read sum at t:

$$\langle E_t, n\rangle = \langle E_{t-1}, n\rangle \ominus \left\langle est_i^{t-1}, w_i^{t-1}\right\rangle \oplus \left\langle est_i^{t-1} + \frac{r_i^t - r_i^{t-1}}{w_i^{t-1}}, w_i^{t-1}\right\rangle =$$

$$= \langle E^{t-1} + \frac{r_i^t - r_i^{t-1}}{n}, n\rangle = \langle A_{t-1} + \frac{r_i^t - r_i^{t-1}}{n}, n\rangle = \langle A_t, n\rangle$$

□

Dynamic Topology

When a link is added, the node adding it just starts to keep track of the messages passed on the link. When a link is removed, the node retroactively cancels the messages that passed through this link, as if it never existed. In both cases, the read average and the estimate average are unchanged.

Lemma 3 (Topology change)**.** *If the invariant $\mathcal{I}$ holds at time $t-1$, and step t is* `addNeighbor` *or* `removeNeighbor`*, then the invariant holds at time t.*

Proof. The `addNeighbor` function does not effect the read sum or the estimate sum. When a link (i, j) failure is discovered by i, the weighted value $sent_i^{t-1}(j)$ is added to the estimate of node i (line 40) and the variable $sent_i^t(j)$ is nullified (line 42). Additionally, the weighted value $received_i^{t-1}(j)$ is subtracted from node i's estimate (line 40), and the variable $received_i^t(j)$ is nullified (line 43). The changes to the node's estimate compensate the nullification of its two summary variables for link (i, j), so the estimate sum at t is the same as at $t-1$. The removal of a link does not change the read sum, so it is equal to its value at $t-1$ as well, and since they were equal at $t-1$, the invariant holds at t. □

Dynamic Node Set

When a node is added, its state is added to the system. When it is removed, its state is removed.

Lemma 4 (Churn)**.** *If the invariant $\mathcal{I}$ holds at time $t-1$, and step t is* `init` *or* `remove`*, then the invariant holds at time t.*

Proof. An addition of a node i with initial estimate r_i^t adds $\langle r_i^t, 1\rangle$ to both the read average and the estimate average (line 8), so they are both equal at step t.

Node i's failure subtracts $\langle r_i^{t-1}, 1\rangle$ from the read sum, and effects the state of node i (only). It removes the node's weighted value, and all of its *received* and *sent* variables. However, notice that node i's weighted value can be partitioned:

$$\langle est_i^{t-1}, w_i^{t-1}\rangle \quad = \quad \langle r_i^{t-1}, 1\rangle \quad \oplus \bigoplus_{j\in neighbors_i^{t-1}} received_i^{t-1}(j) \quad \ominus \bigoplus_{j\in neighbors_i^{t-1}} sent_i^{t-1}(j) \ ,$$

so removing $\langle est_i^{t-1}, w_i^{t-1}\rangle$, $received_i^{t-1}(j)$ and $sent_i^{t-1}(j)$ from the system is equivalent to subtracting $\langle r_i^{t-1}, 1\rangle$ from the previous estimate sum, so $E^t = E^{t-1} - \langle r_i^{t-1}, 1\rangle$. Therefore, the new read sum equals the new estimate sum and the invariant holds as required. □

We conclude by proving Theorem 1, showing by induction that the read average equals the estimate average at all times:

Theorem 1. *In a run of the system, the read average equals the estimate average at all times*

Proof. Since the correctness of the invariant implies the correctness of the claim, we prove that the invariant holds at all times. Initially, at $t = 0$, the invariant holds, since for any node i, the component of the read sum is identical to that of the estimate sum: $\langle r_i^t, 1\rangle = \langle est_i^t, 1\rangle$. At each step t, assuming the invariant holds at $t-1$, one of the 7 possible events occur, and according to Lemmas 1–4 the invariant holds at t as well. □

5.2 Liveness

We show that after GST the estimate at all nodes converges to the read average.

We first note that we can track the propagation from a node i as of time t. To do that, we partition the weighted value of a node k at time $t' > t$ into two components: *prop*, the propagation of i's weighted value at t to k at t', and *agg*, the aggregation from all nodes but i. We define the values of the components by recursion. The weighted value of a node k at time $t' \geq t$ is partitioned to the propagated component $\langle est_i^t, w_{k,prop}^{t'}\rangle$, and the aggregated component $\langle est_{k,agg}^{t'}, w_{k,agg}^{t'}\rangle$. Though these definitions depend on i and t, we fix i and t and omit them, to make the expressions more clean:

$$\langle est_k^{t'}, w_k^{t'}\rangle = \langle est_{k,agg}^{t'}, w_{k,agg}^{t'}\rangle + \langle est_i^t, w_{k,prop}^{t'}\rangle \ .$$

This partitioning is defined for all nodes and for all times starting at t. The *prop* component is called the *component of est_i^t at node k at time t'.*

Initially, at t, at all nodes $k \neq i$, *agg* is the weighted value $\langle est_k^t, w_k^t\rangle$, and *prop* is $\langle 0, 0\rangle$. At node i, *agg* is $\langle 0, 0\rangle$ and *prop* is $\langle est_i^t, w_i^t\rangle$. If the operation at t' is a push send at node k, then

$$\begin{aligned}\langle est_k^{t'}, w_k^{t'}\rangle = \langle est_{k,agg}^{t'}, w_{k,agg}^{t'}\rangle + \langle est_i^t, w_{k,prop}^{t'}\rangle = \\ = \langle est_{k,agg}^{t'-1}, w_{k,agg}^{t'-1}/2\rangle + \langle est_i^t, w_{k,prop}^{t'-1}/2\rangle = \langle est_k^{t'-1}, w_k^{t'-1}/2\rangle \ ,\end{aligned}$$

and the message sent is partitioned $\langle est^{t'-1}_{k,agg}, w^{t'-1}_{k,agg}/2\rangle + \langle est^t_i, w^{t'-1}_{k,prop}/2\rangle$. If the operation at t' is a push receive at node k of a message $\langle v_{\text{in}}, w_{\text{in}}\rangle$ partitioned to $\langle w_{\text{in},agg}, est_{\text{in},agg}\rangle$ and $\langle est^t_i, w_{\text{in},agg}\rangle$ components, then

$$\begin{aligned}\langle est^{t'}_k, w^{t'}_k\rangle = \langle est^{t'}_{k,agg}, w^{t'}_{k,agg}\rangle + \langle est^t_i, w^{t'}_{k,prop}\rangle = \\ = \langle est^{t'-1}_{k,agg}, w^{t'-1}_{k,agg}\rangle + \langle w_{\text{in},agg}, est_{\text{in},agg}\rangle + \langle est^t_i, w^{t'-1}_{k,prop}\rangle + \langle est^t_i, w_{\text{in},prop}\rangle = \\ = \langle est^{t'-1}_k, w^{t'-1}_k\rangle + \langle v_{\text{in}}, w_{\text{in}}\rangle \ .\end{aligned}$$

We define the *ratio* of the est^t_i component at a node k to be the ratio between its weight and the total weight at the node, i.e., $\frac{w_{k,prop}}{w_{k,prop}+w_{k,agg}} = \frac{w_{k,prop}}{w_k}$. We proceed to prove that for any time t larger than GST, eventually all nodes have a component of est^t_i with a ratio that is bounded from below.

Lemma 5. *For any time $t > GST$ and node i, there exists a time $t' > t$ after which every node j has an est^t_i component with ratio larger than $\left(\frac{q}{n}\right)^{n/q}$.*

Proof. Denote by M_s the set of nodes with an est^t_i component. Denote by w_{M_s} the sum of weights in the nodes in M_s, and in push messages sent from nodes in $M_{s'}$ with $s' \leq s$ and not yet received. We show by induction that at any $s > t$, at all nodes with an est^t_i component, the ratio of the est^t_i component is at least $\left(\frac{q}{w_{M_s}}\right)^{w_{M_s}/q}$. Recall that $q < w_{M_s}$, hence $\frac{q}{w_{M_s}} < 1$.

At time t the only node with an est^t_i component is i with a ratio of 1.0, and the invariant holds. Consider the system at time s, assuming the invariant holds at $s-1$. We show that after any of the possible events at $s-1$, the invariant still holds. Note that pull events have no direct effect — they only trigger the push of line 31.

1. <u>PUSH send</u>: No effect on the invariant. The ratio at the sender stays the same, and w_M is unchanged.

2. <u>PUSH receive from $j \notin M$ by $k \notin M$</u>: No effect on the invariant since no nodes in M are concerned.

3. <u>PUSH receive from $j \in M$ by $k \notin M$</u>: Two things change: (1) $w_{M_s} = w_{M_{s-1}} + w^{s-1}_k$ and (2) k becomes a part of M. The first change decreases the lower bound, therefore the assumption holds at s for all nodes in M_{s-1}. Denote by α the ratio at j when it sent the message. According to the induction assumption, $\alpha \geq \left(\frac{q}{w_{M_{s-1}}}\right)^{w_{M_{s-1}}/q}$. The new ratio at k is minimal when the weight of the received message is minimal (i.e., q). Therefore, the ratio at k, which is now also in M, is at least

$$\alpha \cdot \frac{q}{w^{s-1}_k} \overset{\substack{\text{induction}\\ \text{assumption}}}{\geq} \left(\frac{q}{w_{M_{s-1}}}\right)^{w_{M_{s-1}}/q} \frac{q}{w^{s-1}_k} > \left(\frac{q}{w_{M_{s-1}} + w^{s-1}_k}\right)^{\frac{w_{M_{s-1}}+w^{s-1}_k}{q}} = \left(\frac{q}{w_{M_s}}\right)^{w_{M_s}/q} \ .$$

We conclude that the ratio at all the nodes in M_s is larger than the bound at s, therefore the induction assumption holds.

4. PUSH receive from $j \notin M$ by $k \in M$: Denote the weight of the message by w_{in}. Two things change: (1) $w_{M_s} = w_{M_{s-1}} + w_{\text{in}}$ and (2) the ratio at k. The change of w_M decreases the bound, therefore the assumption holds at s for all nodes other than k. The relative weight at k before receiving is at least $\left(\frac{q}{w_{M_{s-1}}}\right)^{w_{M_{s-1}}/q}$. Therefore, after receiving the message, it is at least

$$\left(\frac{q}{w_{M_{s-1}}}\right)^{w_{M_{s-1}}/q} \cdot \frac{q}{q + w_{\text{in}}} > \left(\frac{q}{w_{M_{s-1}} + w_{\text{in}}}\right)^{\frac{w_{M_{s-1}} + w_{\text{in}}}{q}} = \left(\frac{q}{w_{M_s}}\right)^{w_{M_s}/q} .$$

   We conclude that the ratio at all the nodes in M_s is larger than the bound at s, therefore the induction assumption holds.

5. PUSH receive from $j \in M$ by $k \in M$: The ratio doesn't go below the minimum between the ratios in j and k, therefore the invariant's correctness follows directly from the induction assumption.

Once a node has an est_i^t component, it will always have an est_i^t component (no operation removes it), and eventually it will succeed sending a push message to all of its neighbors (fairness). Therefore, due to the connectivity of the network after GST, and according to the induction above, eventually every node has an est_i^t component with a ratio not smaller than $\left(\frac{q}{n}\right)^{n/q}$. □

We proceed to prove the main result of this section.

Theorem 2 (Liveness)**.** *After GST, the estimate error at all nodes converges to zero.*

Proof. We define a series of times $t_0, t_1, t_2, \ldots$ recursively. The initial time is $t_0 = \text{GST}$. For every t_{p-1} we define t_p to be the time at which each node $k \in \mathcal{N}_{\text{GST}}$ has an est_i^{p-1} component with ratio at least $\left(\frac{q}{n}\right)^n$ for each $i \in \mathcal{N}_{\text{GST}}$. Such a t_p exists according to Lemma 5.

Denote by $e_{\max}^{p-1}$ the largest estimate at a node at time t_{p-1}, i.e., $e_{\max}^{p-1} = \max_i\{est_i^{p-1}\}$. Assume without loss of generality that the average is zero. If all node estimates are the exact average, then the estimate is zero at all nodes and it does not change. Otherwise, $e_{\max}^{p-1}$ is strictly positive, and there exists some node j whose estimate is negative. At t_p, a node i has a component of $est_j^{t_{p-1}}$ with weight at least $\left(\frac{q}{n}\right)^{n/q}$ (Lemma 5). The weight of the rest of its components is smaller than n, and their values are at most $e_{\max}^{p-1}$. Therefore, the estimate at i at t_p is bounded:

$$est_i^{t_p} < \left(n \cdot e_{\max}^{p-1} + \left(\frac{q}{n}\right)^{n/q} \cdot est_j^{t_{p-1}}\right) \cdot \frac{1}{n + \left(\frac{q}{n}\right)^{n/q}} \overset{est_j^{t_{p-1}}<0}{<} \frac{n}{n + \left(\frac{q}{n}\right)^{n/q}} e_{\max}^{p-1} .$$

The estimate at i is similarly bounded from below with respect to the minimal value at t_{p-1}. The maximal error (absolute distance from average) at t_p is therefore bounded by $\frac{n}{n+\left(\frac{q}{n}\right)^{n/q}}$ the maximal error at t_{p-1}. We conclude that the maximal error decreases at least exponentially with p, and therefore the estimates converge to the average. □

5.3 Performance

We say the the suffix of a run is *uniform synchronous* if (1) the choice of which node runs and choice of which neighbor it chooses for data exchange is uniformly random, and (2) the latency of all operations and links is 0, so the only thing that takes time in the system is the time between the periodic `send` functions.

We now analyze the convergence rate in uniform synchronous settings. We assume that the network is fully connected (actually this can be relaxed to require a network with good connectivity), the nodes operation frequency is similar, and the neighbor choice is uniform random. We further assume that the network is large enough and well connected, and that the communication pattern is "random enough" so the values are also roughly independent. We show that after GST, in these settings, the MSE converges at an exponential ratio.

Normal-like distribution

The value at each node is the result of adding values from its neighbors. After running long enough, we can assume, as explained above, that this is a sum of independent and identically distributed random variables. As such, the distribution of the node estimates should be normal, according to the central limit theorem. The assumption was confirmed by simulation, using the Jarque-Bera normality test [12].

Exchange gossip

To analyze the algorithm, we begin with the special case of exchange gossip, where in each step two nodes perform push send to each other. With this timing, the weights at all nodes always remain 1.

Let us consider the progress of LiMoSense in these settings. In each step, two nodes send their estimates to each other, and then replace them with their average. Assume the distribution at time t is standard normal, i.e., with mean $\mu = 1$ and variance $\sigma^2 = 1$. At the next step, two values are chosen according to a normal distribution and are merged. We calculate the expected variance after one step of the algorithm at time t'. For all x, y, we multiply the probability density of choosing these values by the resulting variance after their merger. This new variance is the variance of the normal distribution before the merger, i.e., 1, from which we subtract the effect of the two values, each with weight $1/n$, and then add the average twice, i.e., with weight $2/n$. The computation is given in equation 4, showing the expected variance after a step is $1 - \frac{1}{n}$.

$$\int_{-\infty}^{\infty} dx \int_{-\infty}^{\infty} dy \underbrace{\frac{1}{2\pi} e^{\frac{-x^2-y^2}{2}}}_{\text{Probability density of choosing } x \text{ and } y} \underbrace{\left(1 - \frac{1}{n}x^2 - \frac{1}{n}y^2 + \frac{2}{n}\left(\frac{x+y}{2}\right)^2\right)}_{\text{Variance in next step for the given } x,\, y} = 1 - \frac{1}{n} \tag{4}$$

We have conducted simulations to verify the predicted convergence rate of LiMoSense. We simulated a fully connected network of 100 sensors. The samples were taken from a standard normal distribution. Figure 1 shows mean square error of the nodes and the value predicted by the analysis. The simulation value is averaged over 100 instances of the simulation. The result perfectly fits the behavior predicted by Equation 4.

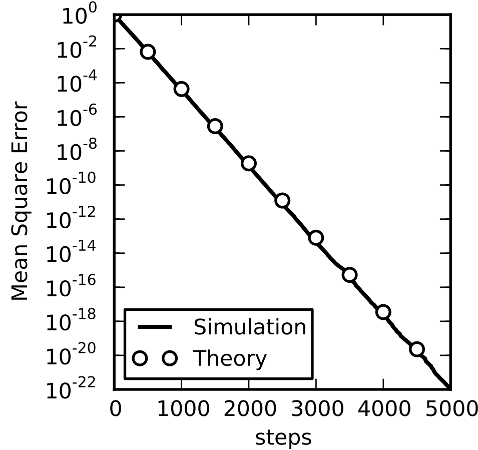


Figure 1: Simulation and theory (Equation 4) of exchange LiMoSense convergence rate in a static fully connected network of 100 nodes, with read values drawn from the standard normal distribution. We see that the algorithm converges exactly as predicted by Equation 4.

Push gossip

A similar computation is given in Equation 5 for push gossip. In this case, the value at x remains unchanged, with half the weight, and the value at y is replaced by the weighted average with the value received from x, with the aggregated weight. The expected new variance in this case is $1-\frac{2}{3n}$, slightly smaller than in the exchange case. Simulation shows that a small correction factor of about 0.9 is required, possibly due to the asymmetric distribution of weights.

$$\int_{-\infty}^{\infty} dx \int_{-\infty}^{\infty} dy \left(\underbrace{\frac{1}{2\pi} e^{\frac{-x^2-y^2}{2}}}_{\substack{\text{Probability density}\\ \text{of choosing } x \text{ and}\\ y}} \cdot \underbrace{\left(1 - \frac{1}{n}x^2 - \frac{1}{n}y^2 + \frac{1}{2n}x^2 + \frac{3}{2n}\left(\frac{\frac{1}{2}x+y}{3/2}\right)^2 \right)}_{\substack{\text{Variance in next step for the given } x,\\ y}} \right) = 1 - \frac{2}{3n} \quad (5)$$

6 Simulation

In this section, we describe simulations we have conducted to evaluate LiMoSense under various dynamic scenarios. Our goal is to asses how fast the algorithm reacts to changes, and succeeds to provide accurate information. We compare LiMoSense to a periodically-restarting Push-Sum algorithm. We explain our methodology and metrics in Section 6.1

We first study how the algorithm copes with different types of data read changes - a gradual "creeping" change of all values, occurring, e.g., when temperature is gradually rising (Section 6.2), an abrupt value change captured by a "step function" (Section 6.3), and a temporary glitch or "impulse" (Section 6.4). We then study the algorithm's robustness to node and link failures (Section 6.5).

6.1 Methodology

We performed the simulations using a custom made python event driven simulation that simulated the underlying network and the nodes' operation. Unless specified otherwise, all simulations are of a network of 100 nodes in a fully connected network. We have seen that in well connected networks, the convergence behavior is similar to that of a fully connected network. The simulation proceeds in steps, where in each step, the topology and read values may change according to the simulated scenario, and one node performs a pull or push action. Scheduling is uniform synchronous, i.e., the node performing the action is chosen uniformly at random.

Unless specified otherwise, each scenario is simulated 1000 times. In all simulations, we track the algorithms' output and accuracy over time. In all of our graphs, the X axis represents steps in the execution. We depict the following three metrics for each scenario:

(a) **base station**. We assume that a base station collects the estimated read average from some arbitrary node. We show the median of the values obtained in the runs at each step.

(b) **ε-inaccuracy**. For a chosen ε, we depict the percentage of nodes whose estimate is off by more than ε after each step. The average of the runs is depicted.

(c) **MSE**. We depict the average square distance between the estimates at all nodes and the read average at each step. The average of all runs is depicted.

We compare LiMoSense to a Push-Sum algorithm that re-starts at a constant frequency — every 5000 steps unless specified otherwise. In base station results, we also show the read average, i.e., the value the algorithms are trying to estimate.

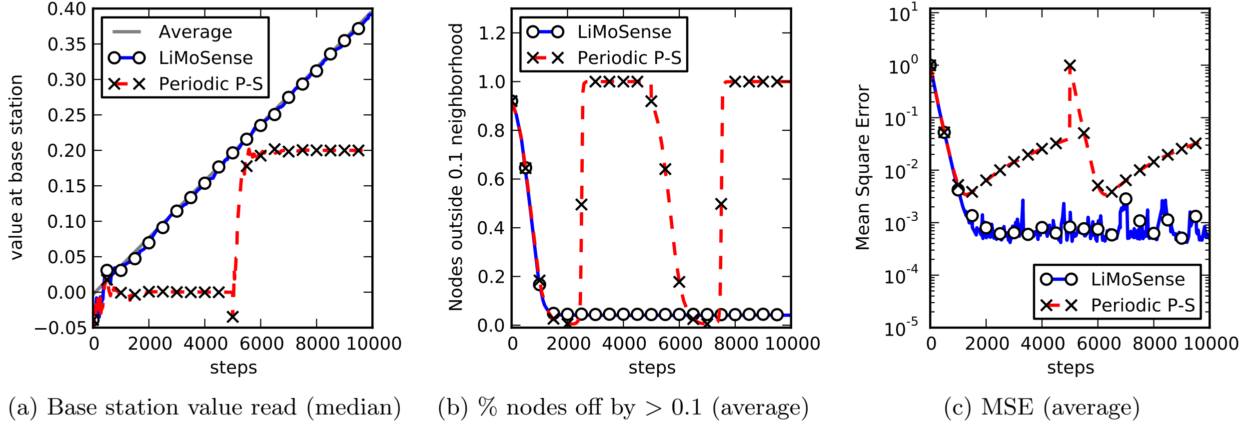
6.2 Creeping Change

This simulation investigates the behavior of the algorithm when the values read by the sensors slowly increase. This may happen if the sensors are measuring rainfall that is slowly increasing. Every 10 steps, a random set of 5 of the nodes read values larger by 0.01 than their previous reads. The initial values are taken from the standard normal distribution. The results are shown in Figure 2.

In Figure 2a we see that the average is increasing at a constant rate, and the LiMoSense base station closely follows. The restarting Push-Sum, however, tries to update its value only at constant intervals, making it impossible for it to follow the read average. Due to its significant error, the time it takes for updating is so long that it never actually reaches the read average line.

In Figure 2b we see that after its initially convergence, the LiMoSense algorithm has most of the nodes maintain a good estimate of the read average. Less than 10% of the nodes have estimates worse than 0.1, whereas the restarting Push-Sum algorithm has no nodes in this neighborhood most of the time, and most of the nodes in the neighborhood for short intervals.

Finally, in Figure 2c we see that the LiMoSense algorithm maintains a small MSE, with some noise, whereas the restarting Push-Sum algorithm's error quickly converges after restart, until the creeping change takes over and dominates the MSE causing a steady increase until the next restart.



(a) Base station value read (median) (b) % nodes off by > 0.1 (average) (c) MSE (average)

Figure 2: **Creeping value change** Every 10 steps, 5 random reads increase by 0.01. We see that LiMoSense promptly tracks the creeping change. It provides accurate estimates to 95% of the nodes, with an MSE of about 10^{-3} throughout the run. In contrast, Periodic Push-Sum is accurate only following restarts.

6.3 Step Function

This simulation investigates the behavior of the algorithm when the values read by some sensors are shifted. This may occur due to a fire outbreak in a limited area, as close-by temperature nodes suddenly read high values.

At step 2500, a random set of 10 nodes read values larger by 10 than their previous reads. The initial values are taken from the standard normal distribution. The results are shown in Figure 3.

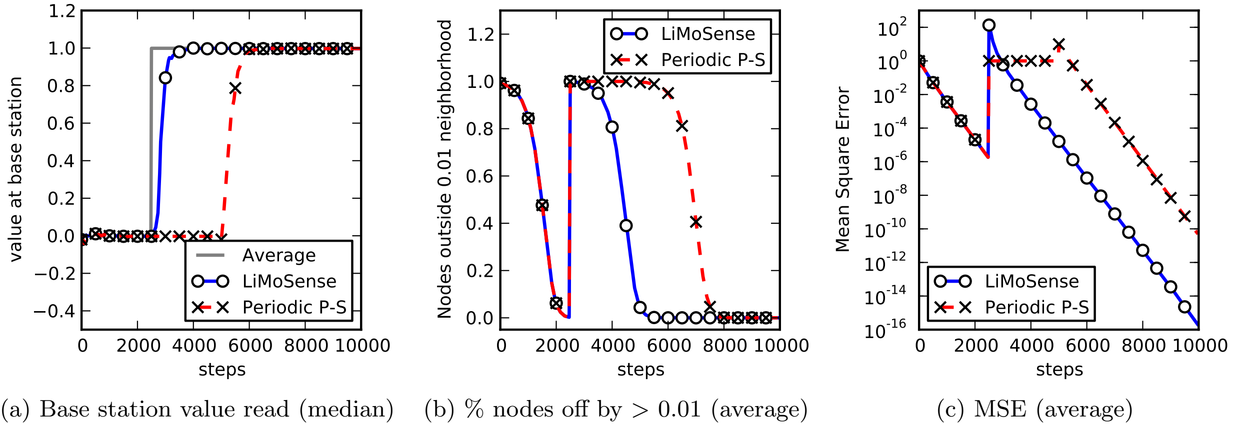
Figure 3a shows how the LiMoSense algorithm updates immediately after the shift, whereas the periodic Push-Sum algorithm updates at its first restart only. Figure 3b shows the ratio of erroneous sensors with error larger than 0.01 quickly dropping after — right after the read average change for LiMoSense, and at restart for the periodic Push-Sum. Figure 3c shows the MSE decrease, as predicted in Section 5.3. Both LiMoSense and periodic Push-Sum converge at the same rate, but start a different times.

6.4 Impulse Function

This simulation investigates the behavior of the algorithm when the reads of some sensors are shifted for a limited time, and then return to their previous values. This may happen due to sensing errors, causing the nodes to read irrelevant data. As an example, one may consider the case of a heavy vehicle driving by seismic sensors used to detect earthquakes. The close-by sensors would read high seismic activity for a short period of time.

At steps 2500 and 6000, a random set of 10 nodes read values larger by 10 than their previous reads, and after 100 of each time they return to their values before the shift. The initial values are taken from the standard normal distribution. The results are shown in Figure 4.

The LiMoSense algorithm's reaction is independent on the impulse time — a short period of noise raises the estimate at the base station as the impulse value propagates from the sensors that read the impulse. Then, once the impulse is canceled, this value decreases. The estimate with



(a) Base station value read (median) (b) % nodes off by > 0.01 (average) (c) MSE (average)

Figure 3: **Response to a step function** At step 2500, 10 random reads increase by 10. We see that LiMoSense immediately reacts, quickly propagating the new values. In contrast, Periodic Push-Sum starts its new convergence only after its restart.

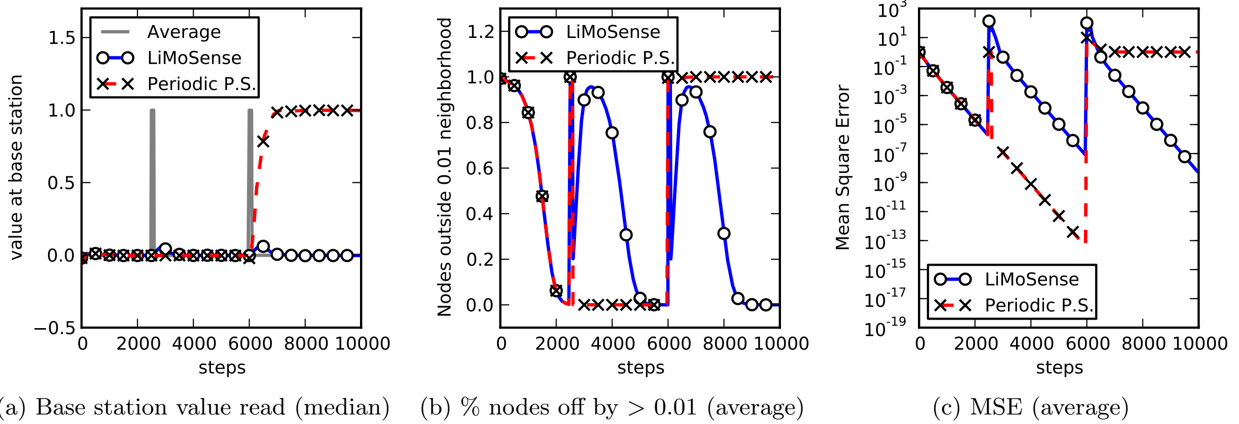
respect to the read average is shown in Figure 4a, and the ratio of correct sensors is in Figure 4b. The impulse essentially restarts the MSE convergence, as shown in Figure 4c — After an impulse ends, the error returns to its starting point and starts convergence anew.

The response of the periodic Push-Sum depends on the time of impulse. If the impulse occurs between restarts (as in step 2500), the algorithm is completely oblivious to it. All three figures 4a–4c show that apart from the impulse time, convergence continues as if it never happened. If, however, a restart occurs during the impulse (as in step 6000), then the impulse is sampled and the algorithm converges to this new value. This convergence is similar to the reaction to the step function of Section 6.3, only in this case it promptly becomes stale as the impulse ends. Figure 4a shows the error quickly propagating to the base station. Since the algorithm has the estimates converge to the read average during impulse, the ratio of inaccurate nodes is 1.0 once the impulse ends, and the MSE stabilizes at a large value as all nodes converge to the wrong estimate.

6.5 Robustness

To investigate the effect of link and node failures, we construct the following scenario. The sensors are spread in the unit square, and they have a transmission range of 0.7 distance units. The neighbors of a sensor are the sensors in its range. The system is run for 3000 steps, at which point, due to battery decay, the transmission range of 10 sensors decreases by 0.99. Due to this decay, the nodes' links with some of their neighbors fail, and they employ their `removeNeighbor` functions. We see the effect of this link removal in Figure 5. In Figure 5a the effect can hardly be seen, but a temporary decrease of the accurate nodes can be seen in Figure 5b, and in Figure 5c we see the MSE rising sharply. The failure of links does not effect the periodic Push-Sum algorithm, which continues to converge.

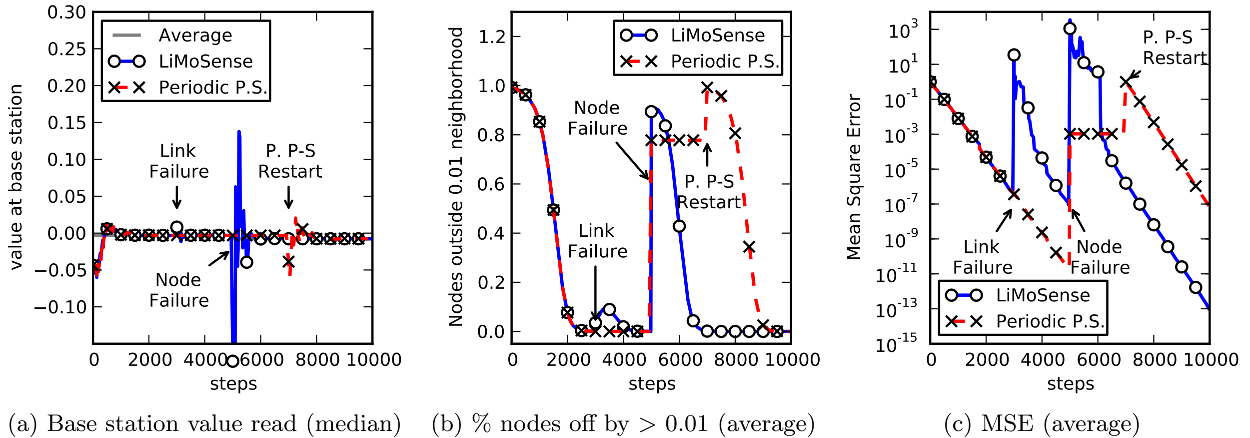
In step 5000, a node fails, removing its read value from the read average. Upon node failure, all its neighbors call their `removeNeighbor` functions. Figure 5a shows the extreme noise at the base station caused by the failure, and in Figure 5b we see the ratio of accurate nodes decreasing sharply before converging again. We see in Figure 5c that the node removal effectively requires



(a) Base station value read (median) (b) % nodes off by > 0.01 (average) (c) MSE (average)

Figure 4: **Response to impulse** At steps 2500 and 6000, 10 random values increase by 10 for 100 steps. Both impulses cause temporary disturbances in the output of LiMoSense. Periodic Push-Sum is oblivious to the first impulse, since it does not react to changes. The restart of Push-Sum occurs during the second impulse, causing it to converge to the value measured then.

the MSE convergence to restart. However, Periodic Push-Sum has no mechanism for reacting to the change until its next restart. Since the average changes, until that time, the percentage of inaccurate nodes sharply rises to 1.0, and the MSE reaches a static value, as the estimates at the nodes converge to the wrong average. Since in every run a different node crashes, and the median of the removed value is 0, the node crash does not effect the median periodic Push-Sum value at the base station in Figure 5a.



(a) Base station value read (median) (b) % nodes off by > 0.01 (average) (c) MSE (average)

Figure 5: **Failure robustness** In a disc graph topology, the radio range of 10 nodes decays in step 3000, resulting in about 7 lost links in the system. Then, in step 5000, a node crashes. Each failure causes a temporary disturbance in the output of LiMoSense. Periodic Push-Sum is oblivious to the link failure. It recovers from the node failure only after the next restart.

7 Conclusion

We presented LiMoSense, a fault-tolerant live monitoring algorithm for dynamic sensor networks. LiMoSense uses gossip to dynamically track and aggregate a large collection of ever-changing sensor reads. It overcomes message loss, node failures and recoveries, and dynamic network topology changes. We have proven correctness of LiMoSense; and illustrated by simulation its ability to quickly react to network and value changes and provide accurate information.

References

[1] G. Asada, M. Dong, T. Lin, F. Newberg, G. Pottie, W. Kaiser, and H. Marcy, "Wireless integrated network sensors: Low power systems on a chip," in *ESSCIRC*, sep 1998, pp. 9–16.

[2] B. Warneke, M. Last, B. Liebowitz, and K. Pister, "Smart dust: communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, Jan 2001.

[3] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-based computation of aggregate information," in *FOCS*, 2003, pp. 482–491.

[4] S. P. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Gossip algorithms: design, analysis and applications," in *INFOCOM*, 2005, pp. 1653–1664.

[5] D. Mosk-Aoyama and D. Shah, "Computing separable functions via gossip," in *PODC*, 2006.

[6] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," in *OSDI*, 2002.

[7] Y. Birk, I. Keidar, L. Liss, and A. Schuster, "Efficient dynamic aggregation," in *DISC*, 2006, pp. 90–104.

[8] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang, "Network imprecision: A new consistency metric for scalable monitoring," in *OSDI*, 2008, pp. 87–102.

[9] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson, "Synopsis diffusion for robust aggregation in sensor networks," in *SenSys*, 2004, pp. 250–262.

[10] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, pp. 182–209, 1985.

[11] P. Jesus, C. Baquero, and P. S. Almeida, "Fault-tolerant aggregation by flow updating," in *DAIS*, 2009, pp. 73–86.

[12] C. M. Jarque and A. K. Bera, "Efficient tests for normality, homoscedasticity and serial independence of regression residuals," *Economics Letters*, vol. 6, no. 3, pp. 255 – 259, 1980.