

LiMoSense – Live Monitoring in Dynamic Sensor Networks

Ittay Eyal, Idit Keidar, and Raphael Rom

Department of Electrical Engineering,
Technion — Israel Institute of Technology
{ittay@tx, idish@ee, rom@ee}.technion.ac.il

Abstract. We present LiMoSense, a fault-tolerant live monitoring algorithm for dynamic sensor networks. This is the first asynchronous robust average aggregation algorithm that performs live monitoring, i.e., it constantly obtains a timely and accurate picture of dynamically changing data. 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 the correctness of LiMoSense; we use simulations to illustrate its ability to quickly react to changes of both the network topology and the sensor reads, and to provide accurate information.

1 Introduction

To perform monitoring of large environments, we can expect to see in years to come sensor networks with thousands of light-weight nodes monitoring conditions like seismic activity, humidity or temperature [2, 14]. 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 nature of these widely spread networks 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 dynamic settings, it is particularly important to perform *live monitoring*, i.e., to constantly obtain a timely and accurate picture of the ever-changing data. However, most previous solutions have focused on a static (single-shot) version of the problem, where the average of a single input-set is calculated [10, 4, 12, 11]. 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. For further details on previous work, see Section 2. In this paper we tackle the problem of live

monitoring in a dynamic sensor network. This problem is particularly challenging due to the dynamic nature of sensor networks, where nodes may fail and may be added on the fly (churn), and the network topology may change due to battery decay or weather change. The formal model and problem definition appear in Section 3.

In Section 4 we present our new **Live Monitoring for Sensor** networks algorithm, LiMoSense. Our algorithm computes the average over a dynamically changing collection of sensor reads. The algorithm has each node calculate an estimate of the average, which continuously converges to the current average. The space complexity at each node is linear in the number of its neighbors, and message complexity is that of the sensed values plus a constant. At its core, LiMoSense employs gossip-based aggregation [10, 12], 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 again difficult, since a node cannot know whether a previous message it had sent over a faulty link has arrived or not.

In Section 5, we review the correctness proof of the algorithm, showing 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. The complete analysis can be found in the technical report [5].

We evaluate the algorithm’s behavior in general (unstable) settings in Section 6. As convergence time is inherently unbounded in asynchronous systems, 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). Furthermore, to demonstrate the effectiveness of LiMoSense in various dynamic scenarios, we present results of extensive simulations, showing its quick reaction to dynamic data read changes and fault tolerance. In order to preserve energy, communication rates may be decreased, and nodes may switch to sleep mode for limited periods. These issues are outside the scope of this work.

In summary, this paper makes the following contributions: (1) It presents LiMoSense, a live monitoring algorithm for highly dynamic and error-prone environments. (2) It proves correctness of the algorithm, namely robustness and eventual convergence. (3) It shows, through analysis and simulation, that LiMoSense converges exponentially fast and demonstrates its efficiency and fault-tolerance in dynamic scenarios.

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 [11, 10, 4, 12]. The only exception we are aware of is work on aggregation with dynamic inputs by Birk et al. [3]; 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 [11]. The hierarchical method incurs considerable resource waste for tree maintenance, and results in aggregation errors in dynamic environments, as shown in [7].

The second approach is gossip-based aggregation at all nodes. To avoid counting the same data multiple times, Nath et al. [13] 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 [6].

An alternative approach to gossip-based aggregation is presented by Kempe et al. [10]. 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]. Jelasity et al. periodically restart the push-sum algorithm to handle dynamic settings, trading off accuracy and bandwidth. 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. Jesus et al. [9, 1] also solve aggregation in dynamic settings, overcoming message loss, dynamic topology and churn. However, they consider synchronous settings, and they do not prove correctness nor analyze the behaviour of their algorithm with dynamic inputs.

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 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. In a step, a node may change its internal state and send messages to its 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 denote¹ 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 was 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).

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

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

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 $R^t \triangleq \frac{1}{|\mathcal{N}_t|} \sum_{i \in \mathcal{N}_t} r_i^t$. 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}_{GST} , the output estimate converges to the read average after GST.

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

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

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[10, 4]: 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.

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

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 . Subtraction operations will be used later, they are denoted by \ominus and are defined below.

$$\langle v_a, w_a \rangle \oplus \langle v_b, w_b \rangle \triangleq \left\langle \frac{v_a w_a + v_b w_b}{w_a + w_b}, w_a + w_b \right\rangle. \quad (1)$$

$$\langle v_a, w_a \rangle \ominus \langle v_b, w_b \rangle \triangleq \langle v_a, w_a \rangle \oplus \langle v_b, -w_b \rangle. \quad (2)$$

Algorithm 1: Failure Free

<pre> 1 state 2 $\langle est_i, w_i \rangle \in \mathbb{R}^2$ 3 $prevRead_i \in \mathbb{R}$ 4 on $init_i(initVal)$ 5 $\langle est_i, w_i \rangle \leftarrow \langle initVal, 1 \rangle$ 6 $prevRead_i \leftarrow initVal$ 7 on $receive_i(\langle v_{in}, w_{in} \rangle)$ <i>from</i> j 8 $\langle est_i, w_i \rangle \leftarrow \langle est_i, w_i \rangle \oplus \langle v_{in}, w_{in} \rangle$ </pre>	<pre> 9 periodically $send_i()$ 10 Choose a neighbor j uniformly at random. 11 $w_i \leftarrow w_i/2$ 12 send $(\langle est_i, w_i \rangle)$ to j 13 on $change_i(newRead)$ 14 $est_i \leftarrow est_i + \frac{1}{w_i} \cdot (newRead - prevRead_i)$ 15 $prevRead_i \leftarrow newRead$ </pre>
--	--

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.

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 10). 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 11–12). When the neighbor receives the message, it merges the accepted weighted value with its own (line 8).

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 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

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 links. There is a rich literature dealing with the means of detecting failures, usually with timeouts. This subject is outside the scope of this work.

Implementing the summary approach naively 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.

The full LiMoSense algorithm, shown as Algorithm 2, is based on the failure-free algorithm. In addition to the state information of the failure-free algorithm, it also maintains the list of its neighbors, and a summary of the data it has sent to and received from 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 to the value of $received_j^i$.

Algorithm 2: LiMoSense

```

1 state
2    $\langle est_i, w_i \rangle \in \mathbb{R}^2$ 
3    $prevRead_i \in \mathbb{R}$ 
4    $neighbors_i \subset \mathbb{N}$ 
5    $sent_i : n \rightarrow (\mathbb{R}^2 \times \mathbb{R}^2) \cup \perp$ 
6    $received_i : n \rightarrow (\mathbb{R}^2 \times \mathbb{R}^2) \cup \perp$ 

7 on  $init_i(initVal)$ 
8    $\langle est_i, w_i \rangle \leftarrow \langle initVal, 1 \rangle$ 
9    $prevRead_i \leftarrow initVal$ 
10   $neighbors_i \leftarrow \emptyset$ 
11   $\forall j : sent_i(j) = \perp$ 
12   $\forall j : received_i(j) = \perp$ 

13 function  $pushSend_i(sendVal)$ 
14   $\langle est_i, w_i \rangle \leftarrow \langle est_i, w_i \rangle \ominus sendVal$ 
15   $sent_i(j) \leftarrow sent_i(j) \oplus sendVal$ 
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   $type \leftarrow$  choose at random from  $\{PUSH, PULL\}$ 
21  if  $type = PUSH$  then
22     $pushSend(\langle est_i, w_i/2 \rangle)$ 
23  else (type = PULL)
24    send  $(\langle est_i, w_i/2 \rangle, PULL)$  to  $j$ 

25 on  $receive_i(\langle v_{in}, w_{in} \rangle, type)$  from  $j$ 
26  if  $type = PUSH$  then
27     $diff \leftarrow \langle v_{in}, w_{in} \rangle \ominus received_i(j)$ 
28     $\langle est_i, w_i \rangle \leftarrow \langle est_i, w_i \rangle \oplus diff$ 
29     $received_i(j) \leftarrow \langle v_{in}, w_{in} \rangle$ 
30  else (type = PULL)
31     $pushSend(\langle v_{in}, -w_{in} \rangle)$ 

32 on  $change_i(r_{new})$ 
33   $est_i \leftarrow est_i + \frac{1}{w_i} \cdot (r_{new} - prevRead_i)$ 
34   $prevRead_i \leftarrow r_{new}$ 

35 on  $addNeighbor_i(j)$ 
36   $neighbors_i \leftarrow neighbors_i \cup \{j\}$ 
37   $sent_i(j) \leftarrow \langle 0, 0 \rangle$ 
38   $received_i(j) \leftarrow \langle 0, 0 \rangle$ 

39 on  $removeNeighbor_i(j)$ 
40   $\langle est_i, w_i \rangle \leftarrow \langle est_i, w_i \rangle \oplus sent_i(j) \ominus received_i(j)$ 
41   $neighbors_i \leftarrow neighbors_i \setminus \{j\}$ 
42   $sent_i(j) \leftarrow \perp$ 
43   $received_i(j) \leftarrow \perp$ 

```

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 an inverse weighted value, and does not change any state variables; these are only changed when the neighbor performs the requested push. The effect of a node pushing a value is equivalent to that of a node pulling (requesting) the inverse value and its neighbor pushing the inverse. Therefore, the use of pull messages does not hamper correctness.

In line 20, the algorithm randomly decides whether to perform push or pull². 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

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

(line 31). Thus, the weights of the *sent* and *received* records fluctuate around 0 rather than grow to infinity. 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 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 *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 Correctness Overview

We defer the correctness proof of LiMoSense to the full version of this paper. We overview here the key theorems.

First, define the invariant \mathcal{I} . 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 *read sum* to be $\langle R^t, n \rangle \triangleq \bigoplus_{i=1}^n \langle r_i^t, 1 \rangle$ and the *estimate sum* to be:

$$\langle E^t, n \rangle \triangleq \bigoplus_{i=1}^n \left(\langle est_i^t, w_i^t \rangle \oplus \bigoplus_{j \in neighbors_i^t} (sent_i^t(j) \ominus received_i^t(j)) \right).$$

The invariant \mathcal{I} states that the estimate sum equals the read sum: $\langle R^t, n \rangle = \langle E^t, n \rangle$.

We prove the following theorem, which states that the invariant is maintained throughout the system's asynchronous operation, despite message loss, topology changes and churn.

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

Then, we prove the following theorem, that shows that after GST the estimates of the nodes eventually mix, i.e., all node estimates converge to the estimate average, which, as the invariant states, equals the read average.

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

6 Evaluation

6.1 static

We say that 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 (negligible with respect to the time between periodic sends). This assumption means that there are no asynchrony issues; it is still weaker than the lock-step assumption often used to evaluate sensor networks.

In uniform synchronous runs, we argue that the nodes' estimates are normally distributed, and it is possible to show analytically that after each push operation, the expected variance decreases by $1 - \frac{1}{n}$. The details of this discussion may be found in the technical report [5].

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 predicted behavior. This result also corresponds to those obtained in [8], where a similar static algorithm is analyzed with the nodes running in lock step.

6.2 Dynamic

In order to evaluate LiMoSense in the dynamic settings it was designed for, we have conducted simulations of various scenarios. Our goal is to assess how fast the algorithm reacts to changes, and succeeds to provide accurate information. Some of the results are described below. Further details can be found in the technical report [5].

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 fully connected network of 100 nodes, with initial values taken from the standard normal distribution. 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.

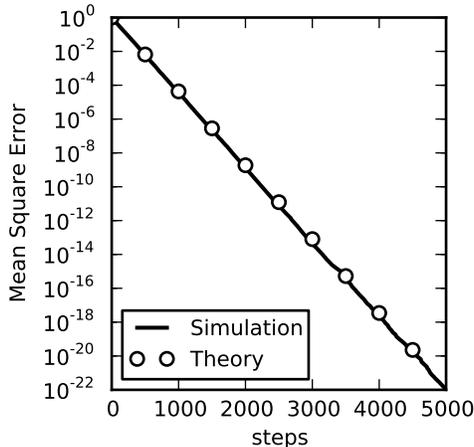


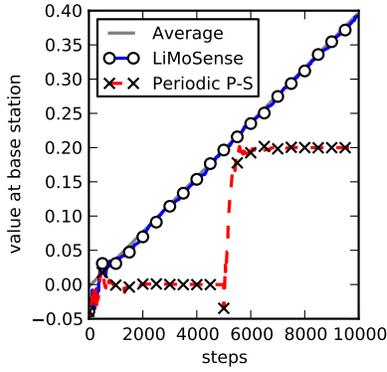
Fig. 1. Exponential convergence rate — Simulation and theory.

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:

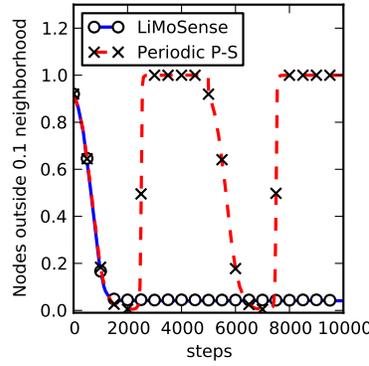
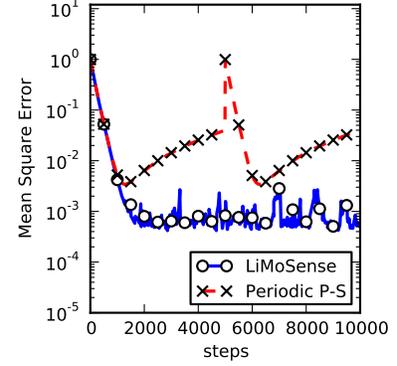
- (a) **base station.** We assume that a base station collects the estimated read average from some arbitrary node. We show the median of the values obtained in the runs at each step.
- (b) **ε -inaccuracy.** For a chosen ε , we depict the percentage of nodes whose estimate is off by more than ε after 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, which does not need restarts, to a Push-Sum algorithm that restarts at a constant frequency — every 5000 steps unless specified otherwise. This number is an arbitrary choice, balancing between convergence accuracy and dynamic response. In base station results, we also show the read average, i.e., the value the algorithms are trying to estimate.

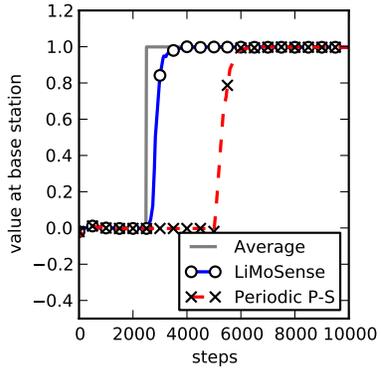
Slow monotonic increase This simulation investigates the behavior of the algorithm when the values read by the sensors slowly increase. This may happen if the sensors are measuring rainfall that is slowly increasing. Every 10 steps, the read values of a random set of 5 nodes increase by 0.01. The results are shown in Figures 2a–2c. LiMoSense closely follows the correct dynamically changing average, whereas a restarting Push-Sum is unable to get close to the moving target.



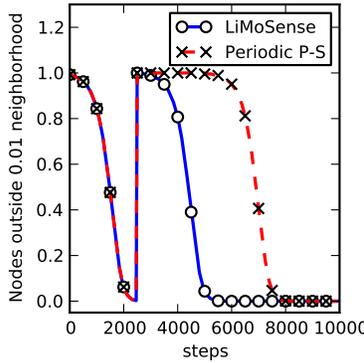
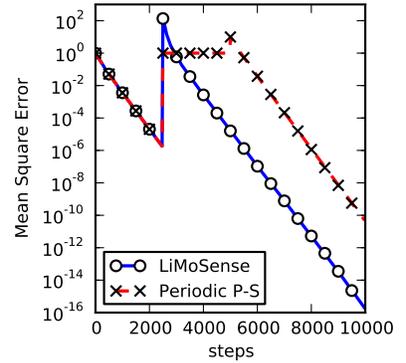
(a) Base station value read (median)

(b) % nodes off by > 0.1 (average)

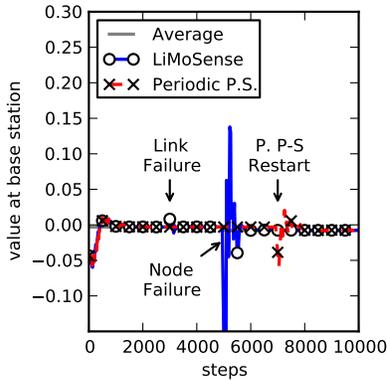
(c) MSE (average)



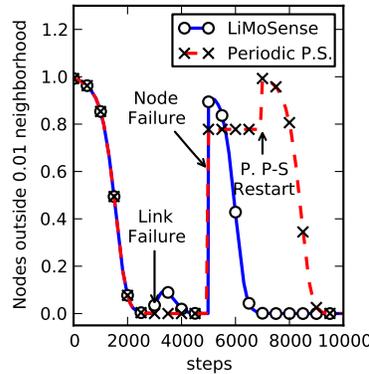
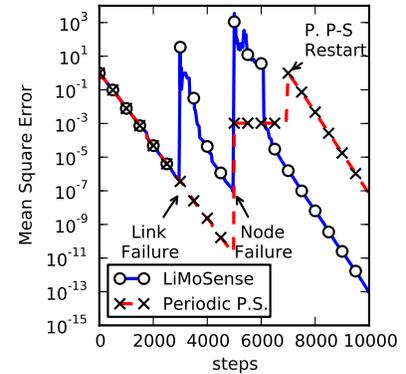
(d) Base station value read (median)

(e) % nodes off by > 0.01 (average)

(f) MSE (average)



(g) Base station value read (median)

(h) % nodes off by > 0.01 (average)

(i) MSE (average)

Fig. 2. (a)–(c) Creeping value change: LiMoSense promptly tracks the creeping change, providing an accurate estimates at 95% of the nodes. **(d)–(f) Response to a step function:** LiMoSense immediately reacts, quickly propagating the new values. **(g)–(i) Failure robustness:** LiMoSense quickly overcomes link loss and node crash.

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, the read values of a random set of 10 nodes increase by 10. The results, shown in Figures 2d–2f, demonstrate how the LiMoSense algorithm updates immediately after the shift, whereas the periodic Push-Sum algorithm updates at its first restart only.

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, about 7 links are lost in the entire system, and the relevant nodes employ their `removeNeighbor` functions. In step 5000, a node fails, removing its read value from the read average. Upon node failure, all its neighbors call their `removeNeighbor` functions.

The results, shown in Figures 2g–2i, shows the small error caused at some of the nodes due to the link failure. A much stronger interruption is caused by the node failure, which actually changes the read average. While the restarting Push-Sum algorithm is oblivious to the link failure, it is unable to recover from the node failure until its next restart.

7 Conclusion

We presented LiMoSense, a fault-tolerant live monitoring algorithm for dynamic sensor networks. This is the first asynchronous robust average aggregation algorithm to accommodate dynamic inputs. LiMoSense employs a hybrid push/pull gossip mechanism 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 the correctness of LiMoSense and illustrated by simulation its ability to quickly react to network and value changes and provide accurate information.

Acknowledgements

This work was partially supported by the Hasso-Plattner Institute for Software Systems Engineering.

References

1. Almeida, P., Baquero, C., Farach-Colton, M., Jesus, P., Mosteiro, M.A.: Fault-tolerant aggregation: Flow updating meets mass distribution. In: OPODIS (2011)
2. Asada, G., Dong, M., Lin, T., Newberg, F., Pottie, G., Kaiser, W., Marcy, H.: Wireless integrated network sensors: Low power systems on a chip. In: ESSCIRC (1998)
3. Birk, Y., Keidar, I., Liss, L., Schuster, A.: Efficient dynamic aggregation. In: DISC (2006)
4. Boyd, S.P., Ghosh, A., Prabhakar, B., Shah, D.: Gossip algorithms: design, analysis and applications. In: INFOCOM (2005)
5. Eyal, I., Keidar, I., Rom, R.: LiMoSense – live monitoring in dynamic sensor networks. Tech. Rep. CCIT 786, Technion, Israel Institute of Technology (2011)
6. Flajolet, P., Martin, G.N.: Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.* 31(2) (1985)
7. Jain, N., Mahajan, P., Kit, D., Yalagandula, P., Dahlin, M., Zhang, Y.: Network imprecision: A new consistency metric for scalable monitoring. In: OSDI (2008)
8. Jelasity, M., Montresor, A., Babaoglu, O.: Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)* 23(3) (2005)
9. Jesus, P., Baquero, C., Almeida, P.: Fault-tolerant aggregation for dynamic networks. In: SRDS (2010)
10. Kempe, D., Dobra, A., Gehrke, J.: Gossip-based computation of aggregate information. In: FOCS (2003)
11. Madden, S., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tag: A tiny aggregation service for ad-hoc sensor networks. In: OSDI (2002)
12. Mosk-Aoyama, D., Shah, D.: Computing separable functions via gossip. In: PODC (2006)
13. Nath, S., Gibbons, P.B., Seshan, S., Anderson, Z.R.: Synopsis diffusion for robust aggregation in sensor networks. In: SenSys (2004)
14. Warneke, B., Last, M., Liebowitz, B., Pister, K.: Smart dust: communicating with a cubic-millimeter computer. *Computer* 34(1) (2001)