Noname manuscript No.
(will be inserted by the editor)

LiMoSense – Live Monitoring in Dynamic Sensor Networks

Ittay Eyal · Idit Keidar · Raphael Rom

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. The algorithm uses a novel technique to bound variable size. We present the algorithm and formally prove its correctness. 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, 19]. 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

The final publication is available at Springer via http://dx.doi.org/10.1007/s00446-014-0213-8. A preliminary version of this paper appears in the proceedings of the 7th International Symposium on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSOR) [7].

This work was partially supported by the Israeli Science Foundation (ISF), Technion Funds for Security Research, the Technion Autonomous Systems Program (TASP), the Intel Collaborative Research Institute for Computational Intelligence (ICRI-CI), and the Hasso-Plattner Institute for Software Systems Engineering (HPI).

I. Eyal
Department of Computer Science, Cornell University, USA.
E-mail: ittay.eyal@cornell.edu

I. Keidar, R. Rom
Department of Electrical Engineering, Technion, Israel.
E-mail: (idish, rom)@ee.technion.ac.il

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 [14, 4, 16, 15]. 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 **Li**ve **Mo**nitoring for **Sens**or 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 [14, 16], 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 prove the correctness 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.

To demonstrate the dynamic behavior of LiMoSense, we present in Section 6 results of simulations of representative scenarios that demonstrate the dynamic reactions. We observe the algorithm's quick reaction to dynamic data read changes and fault tolerance.

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 [15, 14, 4, 16].

For obtaining a single aggregate, two main approaches were employed. The first is hierarchical gathering to a single base station [15]. The hierarchical method

incurs considerable resource waste for tree maintenance, and results in aggregation errors in dynamic environments, as shown in [10].

The second approach is gossip-based aggregation at all nodes. To avoid counting the same data multiple times, Nath et al. [17] 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 [9].

An alternative approach to gossip-based aggregation is presented by Kempe et al. [14]. They introduce Push-Sum, an average aggregation algorithm, and bound its convergence rate, showing that it converges exponentially fast in fully connected networks where nodes operate in lock-step. Fangani and Zampieri [8] analyze the exact convergence rate for a fully connected network, and Boyd et al. [5] analyze this algorithm in an arbitrary topology. Jelasity et al. [12, 11] periodically restart a symmetric version of 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. The aforementioned analyses therefore apply to our algorithm if and when the system stabilizes. Another approach [6] utilizes broadcast to expedite convergence, however unlike our solution it does not allow for message loss, and with a dynamic topology nodes cannot monitor who got their messages.

Wuhib et al. introduce G-GAP [20], a single-shot algorithm for robust aggregation of averages. Their message acknowledge mechanism has similarities to ours, however it addresses a system with much stronger failure detection assumptions: They assume no message loss (that is, a node knows whether its message are received), and no simultaneous crashes.

We are aware of two approaches to aggregate dynamic inputs. The first, by Birk et al. [3], is limited to unrealistic settings, namely a static topology with reliable communication links, failure freedom, and synchronous operation. The second approach, called flow updates (FU) solves aggregation in dynamic settings, overcoming message loss, dynamic topology and churn, albeit in synchronous settings only, running in rounds. In [13, 1], the authors also solve aggregation in dynamic settings, overcoming message loss, dynamic topology and churn. In [13] they provide an empirical evaluation and in [1] they prove correctness and convergence rate for static-input scenarios. However, they consider only 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. Spontaneous steps occur infinitely often. 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 (churn). The set of nodes at time t is denoted $\mathcal{N}^t$ and their number 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 $\mathcal{N}^t$, 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 or removed. We call these *topology events*[2]. 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 We define *global stabilization time*, GST, to be the first time 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 $i \in \mathcal{N}^{\mathrm{GST}}$ 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.

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

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

(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 solves the *Live Average Monitoring Problem* if it 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.

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. This simplified version demonstrates our novel approach for handling dynamic inputs. However, this simplified version is unable to accommodate topology changes, churn, and message loss. To overcome these, we present in Section 4.2 a robust algorithm, in which each node maintains for each of its links a summary of the data communicated over that link thereby enabling it to recover after these changes. These summaries, however, are aggregates of all exchanges on the links, and their size grows, unboundedly. In Section 4.3, we describe the complete LiMoSense algorithm, which also implements a clearing mechanism that results in bounded sizes of all its variables and messages, without resorting to atomicity or synchrony assumptions.

4.1 Failure-Free Dynamic Algorithm

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

The base of the algorithm operates like Push-Sum [14,4]: 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.

In order to accommodate dynamic reads, a node whose read value changes must notify the other nodes. It not only needs to 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.

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 also defined below. The $\oplus$ and $\ominus$ operations are undefined when the sum (resp. difference) between the weights of the operands is zero. We note that

Algorithm 1: Failure-Free Dynamic Algorithm

```
1  state
2      (est_i, w_i) ∈ ℝ^2
3      prevRead_i ∈ ℝ
4  on init_i(initVal)
5      (est_i, w_i) ← (initVal, 1)
6      prevRead_i ← initVal
7  on receive_i((v_in, w_in)) from j
8      (est_i, w_i) ← (est_i, w_i) ⊕ (v_in, w_in)
9  periodically send_i()
10     if w_i ≥ 2q then
11         Choose a neighbor j fairly
12         w_i ← w_i/2
13         send ((est_i, w_i)) to j
14 on change_i(newRead)
15     est_i ← est_i + (1/w_i) · (newRead − prevRead_i)
16     prevRead_i ← newRead
```

the $\oplus$ operation is commutative and both operations are associative.

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

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

The state of a node (lines 2–3) consists of a weighted value, (est_i, w_i), 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 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 $(initVal, 1)$ (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 fairly (line 11). Fairly means that each neighbor is chosen infinitely often. Node i 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 12-13). When the neighbor receives the message, it merges the accepted weighted value with its own (line 8). Nodes keep their weights larger than some small arbitrary size q, by performing a `send` only if the node's weight is larger than $2q$. A small value of q therefore increases the weight sending frequency among nodes, but it does not affect the accuracy of estimation.

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 ever "lost". Second, the algorithm's *convergence* follows from the fact that when a node merges its estimate with that received from a neighbor, the result is closer to the read average than the furthest of the two.

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 shift by the difference between the read values, factored by the relative influence of a single sensor, i.e., $1/n$. So an increase of x increases the system-wide estimate by x/n. However, when a node's read value changes, its estimate has an arbitrary weight of w, so we need to factor the change of its value by $1/w$ to obtain the required system-wide shift. 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 15-16)

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

Note that the value of n^t does not appear in the equation, as it is unknown to any of the nodes.

4.2 Adding Robustness

Overcoming failures is challenging in an asynchronous system, where a node cannot determine whether a message it had sent was successfully received. In order to overcome message loss and link and node failure, each node maintains a summary of its conversations with each of its neighbors. Each node i maintains the aggregates (as weighted sums) of the messages received from and sent to node j in the variables $receivedTotal_i(j)$ and $sentTotal_i(j)$, respectively. Nodes interact by sending and receiving these summaries, rather than weighted values as 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 ever transferred over it. A node failure is treated as the failure of all its links. The resulting algorithm, whose pseudocode is given in Algorithm 2, is robust to message loss, link failure, and churn.

To send, node i adds to $sentTotal_i(j)$ the weighted value it wants to send, and sends $sentTotal_i(j)$ to j (lines 18–20). When receiving this message, node j calculates the newly received weighted value by subtracting its $receivedTotal_i(j)$ variable from the newly received aggregate (line 22). After acting on the received message (line 23), node j replaces its $receivedTotal$ variable with the new weighted value (line 34). 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 most recent message on a link (i, j) is correctly received and there are no messages in transit, the value of $sentTotal_i^j$ is identical to the value of $receivedTotal_j^i$. In order to overcome message

loss, a node i sends its summary to its neighbor j even if its current weight is smaller than $2q$, and the message carries no new information.

Upon notification of topology events, nodes act as follows. When notified of an `addNeighbor` event, a node simply adds the new neighbor to its *neighbors* list (line 29). When notified of a `removeNeighbor` event, a node reacts by nullifying the effect of this link, clearing the state variables, removing the neighbor from its *neighbors* list, and discarding its link records (lines 31–35). Here, the neighbor set of node i may be different on each call to `send`. Fair in this case means that if in infinitely many calls of `send` a node j is a neighbor ($j \in neighbors_i$) infinitely often, then j is chosen infinitely often. The effects of sent and received messages are summarized in the respective *sentTotal* and *receivedTotal* variables. When a node i discovers that link (i, j) failed, it adds the outgoing link summary $sentTotal_i^j$ to its estimate, thus cancelling the effect of ever having sent anything on the link. The incoming link summary, however, is not directly subtracted from the estimate, in order to prevent its weight from becoming negative. Instead, the node adds the incoming link summary $receivedTotal_i^j$ to a buffer – the weighted value $(unrecvVal, unrecvWeight)$. It then lazily subtracts it from its estimate, preserving the estimate weight positive (lines 13–16). The node thus cancels the effect of everything it has received from that neighbor.

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 part of its read value that had propagated through the system is annulled, and it no longer contributes to the system-wide estimate.

4.3 LiMoSense

The summary approach of Algorithm 2 causes summary sizes, namely the weights of $receivedTotal_i(j)$ and $sentTotal_i(j)$, to increase unboundedly as the algorithm progresses. To avoid that, we devise a channel reset mechanism that prevents this without resorting to synchronization assumptions. Instead of storing the aggregates of all received and sent weights, we store only their difference, which can be bounded, and we store the received and sent aggregates only for limited epochs, thereby bounding them as well.

The result is the full LiMoSense algorithm, shown as Algorithms 3a–3b, where the state information of Algorithm 2 is replaced with a more elaborate scheme. Messages are aggregated in epochs, and the aggregate is reset on epoch change. Epochs are defined per node, per link, and per direction, and are identified by binary serial numbers, so each node maintains an incoming and an outgoing serial number per link. Node i maintains for its link with node j the serial numbers $inSN_i(j)$ and $outSN_i(j)$ for the incoming and outgoing weights, respectively. Epochs on different directed links are independent of each other. Neighboring nodes reset their aggregates for their connecting directed link and proceed to the next epoch after reaching consensus on the aggregate values sent in the current epoch. This approach is similar to the classical stop-and-wait message exchange protocol [18]. However, here the receiving end of the link initiates the transition to the next epoch, after receiving multiple messages. Intuitively, the stop-and-wait is performed for the ACKs, each of which acknowledges a set of weight transfers.

Algorithm 2: Robust Dynamic Algorithm with Unbounded State

```
1  state
2      (est_i, w_i) ∈ ℝ²
3      prevRead_i ∈ ℝ
4      neighbors_i ⊂ ℕ, initially ∅
5      sentTotal_i : ℕ → ℝ², initially ∀j : sentTotal_i(j) = (0,0)
6      receivedTotal_i : ℕ → ℝ², initially ∀j : receivedTotal_i(j) = (0,0)
7      (unrecvVal_i, unrecvWeight_i) ∈ ℝ², initially (0,0)
8  on init_i(initVal)
9      (est_i, w_i) ← (initVal, 1)
10     prevRead_i ← initVal
11 periodically send_i()
12     Choose a neighbor j fairly
13     if w_i ≥ 2q then
14         w_toUnrecv ← min(unrecvWeight_i, w_i − q)
15         (est_i, w_i) ← (est_i, w_i) ⊖ (unrecvVal_i, w_toUnrecv)
16         unrecvWeight_i ← unrecvWeight_i − w_toUnrecv
17     if w_i >= 2q then
18         sentTotal_i(j) ← sentTotal_i(j) ⊕ (est_i, w_i/2)
19         (est_i, w_i) ← (est_i, w_i/2)
20     send sentTotal_i(j) to j
21 on receive_i(v_in, w_in) from j
22     diff ← (v_in, w_in) ⊖ receivedTotal_i(j)
23     (est_i, w_i) ← (est_i, w_i) ⊕ diff
24     receivedTotal_i(j) ← (v_in, w_in)
25 on change_i(r_new)
26     est_i ← est_i + (1/w_i) · (r_new − prevRead_i)
27     prevRead_i ← r_new
28 on addNeighbor_i(j)
29     neighbors_i ← neighbors_i ∪ {j}
30 on removeNeighbor_i(j)
31     (est_i, w_i) ← (est_i, w_i) ⊕ sentTotal_i(j)
32     (unrecvVal, unrecvWeight) ← (unrecvVal, unrecvWeight) ⊕ receivedTotal_i(j)
33     neighbors_i ← neighbors_i \ {j}
34     sentTotal_i(j) ← (0,0)
35     receivedTotal_i(j) ← (0,0)
```

For a link (i, j), node i maintains in $sent_i(j)$ and $received_i(j)$ the aggregate sent and received values in the current epoch (rather than the entire history as in the failure-free algorithm). In addition, it maintains in $totalDiff_i(j)$ the difference between the sent and received aggregates over the entire history.

A channel reset for a link (i, j) is initiated by the receiver j when it notices that the weight in $received_i(j)$ reaches a bound `bound` (line 35). This is an arbitrary large bound, for example n, of the weight stored. Node j then (1) increments modulo 2 the serial number on that link, (2) adds the aggregate received values in the completed epoch to its $totalDiff_i(j)$ summary of the link, and (3) clears the aggregate by storing $received_i(j)$ in $cleared_i(j)$ and setting $received_i(j)$ to zero (lines 36–38). Node j will not accept future messages for the previous serial number — it will simply ignore them. On its next send to i (in the inverse direction), node j's message will update i of the epoch reset by sending the index and final aggregate of the completed epoch (line 25).

Algorithm 3a: LiMoSense – part 1

```
1  state
2      (est_i, w_i) ∈ ℝ²
3      prevRead_i ∈ ℝ
4      neighbors_i ⊂ ℕ, initially ∅
5      totalDiff_i : ℕ → ℝ², initially ∀j : totalDiff_i(j) = (0,0)
6      (unrecvVal_i, unrecvWeight_i) ∈ ℝ², initially (0,0)
7      sent_i : ℕ → ℝ², initially ∀j : sent_i(j) = (0,0)
8      outSN_i : ℕ → {0,1}, initially ∀j : outSN_i(j) = (0,0)
                                       (Serial number of outgoing messages)
9      received_i : ℕ → ℝ², initially ∀j : received_i(j) = (0,0)
10     inSN_i : ℕ → {0,1}, initially ∀j : inSN_i(j) = 0
                              (Expected serial number of incoming messages)
11     cleared_i : ℕ → ℝ², initially ∀j : cleared_i(j) = (0,0)
                               (Weight received with previous serial number)
12 on init_i(initVal)
13     (est_i, w_i) ← (initVal, 1)
14     prevRead_i ← initVal
15 periodically send_i()
16     Choose a neighbor j fairly
17     if w_i >= 2q then
18         w_toUnrecv ← min(unrecvWeight_i, w_i − q)
19         (est_i, w_i) ← (est_i, w_i) ⊖ (unrecvVal_i, w_toUnrecv)
20         unrecvWeight_i ← unrecvWeight_i − w_toUnrecv
21     if w_i >= 2q and weight of totalDiff_i(j) > −2 · bound and
       weight of sent_i(j) < 2 · bound then
22         sent_i(j) ← sent_i(j) ⊕ (est_i, w_i/2)  (sentTotal_i(j) ← sentTotal_i(j) + (est_i, w_i/2))
23         totalDiff_i(j) ← totalDiff_i(j) ⊖ (est_i, w_i/2)
24         (est_i, w_i) ← (est_i, w_i/2)
25     send (sent_i(j), outSN_i(j), inSN_i(j) − 1 mod 2, cleared_i(j)) to j
                              (Ack cleared vals and serial of previous epoch)
26 on receive_i((v_in, w_in), msgSN, clearSN, clearVal) from j
27     if clearSN = outSN_i(j) then                          (Relevant clear)
28         outSN_i(j) ← outSN_i(j) + 1 mod 2
29         sent_i(j) ← sent_i(j) ⊖ clearVal   (sentCleared_i(j) ← sentCleared_i(j) + clearVal)
30     if msgSN = inSN_i(j) then                           (Relevant message)
31         diff ← (v_in, w_in) ⊖ received_i(j)
32         (est_i, w_i) ← (est_i, w_i) ⊕ diff
33         totalDiff_i(j) ← totalDiff_i(j) ⊕ diff
34         received_i(j) ← (v_in, w_in)     (receivedTotal_i(j) ← receivedTotal_i(j) + diff)
35         if (weight of received_i(j)) > bound then          (Reset the channel)
36             inSN_i(j) ← inSN_i(j) + 1 mod 2
37             cleared_i(j) ← received_i(j)
                      (receivedCleared_i(j) ← receivedCleared_i(j) + received_i(j))
38             received_i(j) ← (0,0)
```

When notified of the channel reset, the sender resets the aggregate for that channel, and increases modulo 2 the serial number as well. Note that i may have sent messages with the old serial number after the receiver reset the link, but these messages are ignored by j. To prevent this weight from being lost, node i does not reset its aggregate to zero, but rather to the aggregate of messages sent with the old serial number but not cleared (line 29).

Algorithm 3b: LiMoSense – part 2

```
39 on change_i(r_new)
40     est_i ← est_i + (1/w_i) · (r_new − prevRead_i)
41     prevRead_i ← r_new
42 on addNeighbor_i(j)
43     neighbors_i ← neighbors_i ∪ {j}
44     inSN_i(j) ← 0
45     outSN_i(j) ← 0
46 on removeNeighbor_i(j)
47     if Weight of totalDiff_i(j) < 0 then
48         (est_i, w_i) ← (est_i, w_i) ⊖ totalDiff_i(j)
49     else
50         (unrecvVal_i, unrecvWeight_i) ← (unrecvVal_i, unrecvWeight_i) ⊕ totalDiff_i(j)
51     neighbors_i ← neighbors_i \ {j}
52     totalDiff_i(j) ← (0,0)        (receivedCleared_i(j) ← (0,0), sentCleared_i(j) ← (0,0))
53     sent_i(j) ← (0,0)
54     received_i(j) ← (0,0)
55     cleared_i(j) ← (0,0)
```

Upon notification of topology events, nodes act as follows. When notified of an `addNeighbor` event, a node adds the new neighbor to its *neighbors* list and resets the epoch serial numbers for the link (lines 43–45). When notified of a `removeNeighbor` event, a node removes the neighbor from its *neighbors* list and discards its link records. Additionally, it subtracts *totalDiff* from its estimate, thus cancelling the effect of ever having communicated over the link. Unlike Algorithm 2, we cannot separate here the sent from the received weights. To prevent the estimate weight from being negative, we check if the weight in *totalDiff* is positive. If it is, we add it to an aggregate buffer $(\mathit{unrecvVal}, \mathit{unrecvWeight})$, which is later subtracted in stages from *totalDiff* (on send events), as before.

We follow in comments the behavior of four virtual variables, the total sent and received aggregates in $\mathit{sentTotal}_i(j)$ and $\mathit{receivedTotal}_i(j)$, respectively, and the aggregates of everything that was ever cleared from $\mathit{sent}_i(j)$ and $\mathit{received}_i(j)$ in $\mathit{sentCleared}_i(j)$ and $\mathit{receivedCleared}_i(j)$, respectively. These virtual variables all grow unboundedly as the algorithm progresses and we will use them for proving correctness in Section 5.

5 Correctness

In this section, we show that the LiMoSense algorithm (Algorithms 3a–3b) adapts to network topology as well as value changes and converges to the correct average. We start in Section 5.1 by proving that when there are no half-up links, a combination of the system's variables equals the read sum. Then, in Section 5.2, we prove that after GST the estimates at all nodes eventually converge to the average of the latest read values.

5.1 Invariant

We denote by (R^t, n^t) the *read sum* at time t, as shown in Equation 3.

$$(R^t, n^t) = \bigoplus_{i=1}^{n} (r_i^t, 1) \tag{3}$$

We denote by (E^t, n) the weighted sum over all nodes at time t of their (1) weighted values, (2) outgoing link summaries in their *sent* variables, (3) the inverse of their incoming summaries in their *received* variables, and (4) the latest *cleared* received aggregate, if their neighbor has not yet received the reset message. The sum is shown in Equation 4

$$(E^t, n^t) = \bigoplus_{i=1}^{n} \left(\begin{array}{l} (est_i^t, w_i^t) \ominus (unrecvVal_i^t, unrecvWeight_i^t) \oplus \\ \oplus \bigoplus_{j \in neighbors_i^t} \Big(sent_i^t(j) \ominus received_i^t(j) \Big) \ominus \bigoplus_{\substack{j \in neighbors_i^t \text{ s.t.} \\ inSN_i^t(j) \neq outSN_j^t(i)}} cleared_i^t(j) \end{array} \right) \tag{4}$$

We show that if there are no half-up links in the system (each link is known to be up or down by both its nodes), then $R^t = E^t$.

Lemma 1 *For any time t, if for any nodes i and j, either*

$$j \in neighbors_i^t \wedge i \in neighbors_j^t$$

or

$$j \notin neighbors_i^t \wedge i \notin neighbors_j^t \ ,$$

then $R^t = E^t$.

We begin by analyzing the effect of communication steps, then of dynamic steps, and then conclude by proving the statement.

Static Behavior

First, we consider `send` and `receive` events. Note that message loss is not an event and does not affect the state of the system. In particular, it does not affect the correctness of this lemma or the following ones.

Lemma 2 (Static operations) *If step t is either* `send` *or* `receive` *at some node i, then $R^t - E^t = R^{t-1} - E^{t-1}$.*

Proof First, consider a `send` step. If the weight in i is below the threshold of $2q$, no variables are changed (lines 17 and 21), so the lemma trivially holds. If the weight is above the threshold, then a certain weight is subtracted from the pairs $(unrecvVal_i^{t-1}, unrecvWeight_i^{t-1})$ and (est_i^{t-1}, w_i^{t-1}) (lines 18– 20). Since the two pairs appear with opposite signs in Equation 4, the value of E is unchanged. If the weight in i is still above the threshold of $2q$, then the weighted value $(est_i^{t-1}, \frac{1}{2}w_i^{t-1})$ is subtracted from the weighted value of node i, and added to $sent_i^j$, again leaving (E^t, n) according to Equation 4 unchanged.

Next consider a `receive` step. Lines 27–29 handle the outgoing link to j. If the message's *clearSN* is the same as the current $outSN_i(j)$ (line 27), it causes a

reset. Node i reacts by resetting $sent_i(j)$ and incrementing $outSN_i(j)$. Incrementing $outSN_i(j)$ makes it equal to its counterpart $inSN_j(i)$, removing the negative $cleared_j(i)$ element from the sum in Equation 4. Decreasing $sent_i(j)$ by the same value, leaves E^t in Equation 4 unchanged.

Next, if the incoming message carries the appropriate serial number, the incoming value (deducting the previously received value from the incoming aggregate) is added to the weighted value of node j, and the same weighted value is added to $received_j^i$. Since the latter is subtracted in Equation 4, this leaves (E^t, n) unchanged.

Finally, if the weight in the incoming message is too high, the receiver initiates a channel reset. Note that the incoming message serial number equals $outSN_j(i)$. The node increments $inSN_i(j)$, causing $cleared_i(j)$ to be counted in the sum, since after the change it becomes different than $outSN_j(i)$. Then it stores the value of $received_i(j)$ in $cleared_i(j)$, and nullifies $received_i(j)$, both with negative sign in Equation 4, leaving (E^t, n) unchanged.

None of these events changes the read sum, therefore, since neither the read sum nor (E^t, n) change, $R^t - E^t = R^{t-1} - E^{t-1}$

Dynamic Values

When the value read by node i changes from r_i^{t-1} to r_i^t, the node updates its estimate in a manner that changes (E, n) correctly, as shown in the following lemma.

Lemma 3 (Read value change) *If step t is* `change` *at node i, then*

$$R^t - E^t = R^{t-1} - E^{t-1}$$

Proof After the change of the read value, the new read average is $R^t = R^{t-1} + \frac{r_i^t - r_i^{t-1}}{n^{t-1}}$, and the weighted value[3] $\left(est_i^{t-1} + \frac{r_i^t - r_i^{t-1}}{w_i}, w_i\right)$ replaces the weighted value of node i. We show that the new (E, n) changes just like the read sum:

$$\begin{aligned}(E^t, n^t) = (E^{t-1}, n^{t-1}) \ominus \left(est_i^{t-1}, w_i^{t-1}\right) \oplus \\ \oplus \left(est_i^{t-1} + \frac{r_i^t - r_i^{t-1}}{w_i^{t-1}}, w_i^{t-1}\right) = \\ = (E^{t-1} + \frac{r_i^t - r_i^{t-1}}{n^{t-1}}, n^{t-1}) \ ,\end{aligned}$$

leaving the difference between R and E unchanged.

Dynamic Topology

When a link is added, the node adding it 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, both E^t and R^t are unchanged, as we now show.

[3] Note that the weight at a node never drops below q, so the expression is valid.

Lemma 4 (Dynamic Topology) *If step t is* `addNeighbor` *at node i, then $R^t - E^t = R^{t-1} - E^{t-1}$, and if the link between nodes i and j fails and its nodes receive* `removeNeighbor` *at times t_i and t_j (respectively), with $t_i < t_j$, then*

$$(E^{t_i}, n^{t_i}) - (E^{t_i-1}, n^{t_i-1}) = (E^{t_j-1}, n^{t_j-1}) - (E^{t_j}, n^{t_j}) \ .$$

Proof The `addNeighbor` function does not affect the read sum or E^t, so the claim holds. We proceed to handle link failure. When the failure is discovered at t_i by i, the weighted value $totalDiff_i^{t-1}(j)$ is subtracted from est_i or added to $(unrecvVal_i, unrecvWeight_i)$ at node i, and the variables $sent_i(j)$, $received_i(j)$ and $cleared_i(j)$ are nullified. The same happens in j at t_j.

We note that $totalDiff_i(j)$ does not directly appear in Equation 4. We decompose $totalDiff_i(j)$ to the difference between the virtual variables $receivedTotal_i(j)$ and $sentTotal_i(j)$, defined above.

$$totalDiff_i(j) = receivedTotal_i(j) \ominus sentTotal_i(j) \ .$$

We also note that summing virtual variables $sentCleared_i(j)$ and $receivedCleared_i(j)$, together with the real variables $sent_i(j)$ and $received_i(j)$ (respectively) results in $sentTotal_i(j)$ and $receivedTotal_i(j)$, respectively. Therefore, when subtracting $totalDiff_i(j)$ in i's side, we subtract

$$\begin{aligned} totalDiff_i(j) = receivedTotal_i(j) \ominus sentTotal_i(j) = \\ = receivedCleared_i(j) \oplus received_i(j) \ominus \\ \ominus sentCleared_i(j) \ominus sent_i(j) \ . \end{aligned}$$

Now, the $received_i(j)$ and $sent_i(j)$ cancel each other on i's side, as they are subtracted and added (respectively) directly (lines 53–54) when clearing $totalDiff_i(j)$ (line 52).

On the other hand, $receivedCleared_i(j)$ and $sentCleared_i(j)$ must be canceled by an inverse change on j's side. Note that if $inSN_i(j) = outSN_j(i)$, we have $receivedCleared_i(j) = sentCleared_j(i)$, and $cleared_i(j)$ is not counted in Equation 4, whereas if $inSN_i(j) \neq outSN_j(i)$ then $cleared_i(j)$ is counted, and we have $receivedCleared_i(j) + cleared_i(j) = sentCleared_j(i)$. In both cases, the change is canceled, i.e., inverse weighted values are subtracted from/added to est and $(unrecvVal, unrecvWeight)$ (respectively) at t_i and t_j, and the equation in the lemma holds.

Dynamic Node Set

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

Lemma 5 (Dynamic Node Set) *If step t is* `init` *or* `remove`*, then*

$$R^t - E^t = R^{t-1} - E^{t-1}$$

Proof An addition of a node i with initial estimate r_i^t results in $(R^t, n^t) = (R^{t-1}, n^{t-1}) \oplus (r_i^t, 1)$ and $(E^t, n^t) = (E^{t-1}, n^{t-1}) \oplus (r_i^t, 1)$, so their difference is unchanged at step t.

We model the failure of a node i as the failure of all its links, followed by its removal from the system. The failure of the links leaves i with its most recent read value and a weight of one, $(r_i^{t-1}, 1)$, and all other state variables empty ($totalDiff_i(j)$, $sent_i(j)$, etc.), with (E^t, n) unchanged.

Removing the node thus results in $(R^t, n^t) = (R^{t-1}, n^{t-1}) \ominus (r_i^{t-1}, 1)$ and $(E^t, n^t) = (E^{t-1}, n^{t-1}) \ominus (r_i^{t-1}, 1)$, so their difference is unchanged at step t.

We are now ready to prove Lemma 1.

Proof Initially, at $t = 0$, the claim holds, since for any node i, the component of the read sum is identical to that of E^t: $(r_i^t, 1) = (est_i^t, 1)$.

According to Lemmas 2–5, the difference between R^t and E^t changes only due to link failure events. Since there are no half-up links, then if a node i detected the failure of its link with j before t, then j has also detected the failure of the link before t. Lemma 4 shows that the resulting operations by i and j compensate each other, resulting in the required equality at t.

5.2 Convergence

We show that after GST the estimate at all nodes converges to the read average. Since after GST messages are not lost, we can simplify our proof by abstracting away the fact that messages contain aggregated values; instead, we consider each message to deliver only the delta from the previous one, as translated in the code to the *diff* variable upon receipt (line 31).

First, we prove in Section 5.2.1 that connected nodes send each other values infinitely often. Then, in Section 5.2.2, we define the tracking of the propagation of the weighted value from a node i at time t at any time later time. We proceed to show in Section 5.2.3 that there exists a time t' after t such that the ratio of the weight propagated to any node j from any node i, relative to the total weight at j, is bounded from below. In Section 5.2.4 we construct a series of such times, where in each time t_x the values from t_{x-1} have propagated and match this bound. This allows us to prove convergence, as required.

5.2.1 Fair Scheduling

We begin by proving the following lemma.

Lemma 6 *every node sends weight to each of its neighbors infinitely often.*

Proof We prove by contradiction. Assume that a node i never sends a message to its neighbor j. Since neighbors are chosen fairly, i.e., each neighbor is chosen infinitely often, this means the condition of line 21 evaluates to false.

The last part of the condition may evaluate to false only if weight was sent to j but not received. Once this weight is received, node j changes epochs (line 36–38), which will reset $sent_i(j)$ once the next message arrives from j to i, and the condition will evaluate to true.

Therefore, either the first or the second part of the condition do not hold. Assume first that the first part does not hold, i.e., the weight in i is always smaller than $2q$. This means that none of i's neighbors ever sends it weight (from

some time). Otherwise, eventually $unrecvWeight_i$ would drop to zero, and subsequently est_i would rise above zero. Assume that none of i's neighbors ever sends it weight also due to their weights being smaller than $2q$, and continue similarly, i.e., all nodes in the system hold a weight smaller than $2q$. Since the entire weight in the *est* variables is at least n (possibly more, if nodes have non-zero *unrecvWeight* variables)), at least one node must hold a weight larger than one, i.e., larger than $2q$, and we reach a contradiction.

Maintaining our initial assumption, we conclude that there exists some node i that never sends weight to a neighbor j since the second part of the condition holds, i.e., it already sent to j much more than it got back. Once a message from i successfully reaches j, the value of j's $totalDiff_j(i)$ is correctly updated to $-totalDiff_i(j)$, so it is positive. Therefore the second part of the condition is true in j. However, if j sends weight to i, the value of $totalDiff_i(j)$ eventually becomes positive, contradicting our assumption, and we conclude that j stops sending weight at some point, since its weight never rises above $2q$. For that to happen, j must not receive weight from any of its neighbors. So each of j's neighbors either has a weight less than $2q$, or has already sent to j more than it got back. If all of j's neighbors (including i) have sent it more than they got back, than j's weight would be more than $2q$, which we already ruled out. Therefore at least one neighbor k received from j more than it sent, but it does not send weight to j because its own (k's) weight is too small. Now, the same logic that held for j holds for k, and we continue this, forming a chain of nodes. Each node in the chain holds a weight smaller than $2q$. At the end of such a chain (and there is an end, since the number of nodes is finite) there is a node z that does not send weight to any of its neighbors, but has received from each of them more weight than it has sent. The weight at node z is therefore larger than one, and its *totalDiff* for all its neighbors is positive, So the condition in line 21 holds, and it should have sent weights to its neighbors, leading to a contradiction.

After GST, no links failures are detected. Since weights are sent infinitely often between neighbors by Lemma 6, we conclude that there exists a time $\overline{\text{GST}} \geq \text{GST}$ after which the *unrecvWeight* variables at all nodes are zero:

Definition 1 ($\overline{\textbf{GST}}$) The time $\overline{\text{GST}}$ is a time after which for all $i \in \mathcal{N}^{\text{GST}}$ and for all $t > \overline{\text{GST}}$: $unrecvWeight_i^t = 0$.

5.2.2 Propagation Tracking

We explain how to track the propagation of the weighted value from a node i as of some time $t > \overline{\text{GST}}$. The definition recursively defines two components maintained at each node k: The *prop* component, $(est_i^t, w_{k,prop}^{t'})$, which is the propagation of i's weighted value at t to k at t', and the *agg* component, $(est_{k,agg}^{t'}, w_{k,agg}^{t'})$, which is the aggregation from all nodes but i. The *prop* component is called the *component of* est_i^t *at node* k *at time* t'. Though these definitions depend on i and t, we fix i and t and omit them, to make the expressions cleaner.

Definition 2 (Propagation tracking) Initially, at t, at all nodes $k \neq i$, *agg* is the weighted value (est_k^t, w_k^t), and *prop* is $(0, 0)$. At node i, *agg* is $(0, 0)$ and *prop* is (est_i^t, w_i^t).

For all steps $t' > t$:

1. If the operation at t' is a send at node k, then

$$(est^{t'}_{k,agg}, w^{t'}_{k,agg}) = (est^{t'-1}_{k,agg}, w^{t'-1}_{k,agg}/2)$$

and

$$(est^{t}_{i}, w^{t'}_{k,prop}) = (est^{t}_{i}, w^{t'-1}_{k,prop}/2)$$

and the message sent is partitioned:

$$(est^{t'-1}_{k,agg}, w^{t'-1}_{k,agg}/2) \oplus (est^{t}_{i}, w^{t'-1}_{k,prop}/2) \ .$$

2. If the operation at t' is a receive at node k of a message $(v_{\text{in}}, w_{\text{in}})$ partitioned to $(est_{\text{in},agg}, w_{\text{in},agg})$ and $(est^t_i, w_{\text{in},prop})$ components, then

$$(est^{t'}_{k,agg}, w^{t'}_{k,agg}) = (est^{t'-1}_{k,agg}, w^{t'-1}_{k,agg}) \oplus (est_{\text{in},agg}, w_{\text{in},agg})$$

and

$$(est^{t}_{i}, w^{t'}_{k,prop}) = (est^{t}_{i}, w^{t'-1}_{k,prop}) \oplus (est^{t}_{i}, w_{\text{in},prop}) \ .$$

It can be readily seen that the *agg* and *prop* components partition the weighted value at the node k at all times $t' \geq t$:

$$(est^{t'}_{k}, w^{t'}_{k}) = (est^{t'}_{k,agg}, w^{t'}_{k,agg}) \oplus (est^{t}_{i}, w^{t'}_{k,prop}) \ .$$

We define the *component ratio* of node i at a node k to be the ratio between i's *prop* component in k and the total weight at k:

Definition 3 (Component ratio) The *component ratio* of est^t_i at node k at $t' > t$ is

$$\frac{w^{t'}_{k,prop}}{w^{t'}_{k,prop} + w^{t'}_{k,agg}} = \frac{w^{t'}_{k,prop}}{w^{t'}_{k}} \ .$$

5.2.3 Bounded Ratio

We proceed to prove that for any time t after $\overline{\text{GST}}$, eventually each node has a component of est^t_i with a ratio that is bounded from below.

Denote by M^s_i the set of nodes with an est^t_i component at time $s > t$. Denote by $w_{M^s_i}$ the sum of weights at the nodes in M^s_i and in messages sent from nodes in $M^{s'}_i$ with $s' < s$ and not yet received.

Lemma 7 *Given two times s and t s.t. $s > t > \overline{GST}$, at all nodes in M^i_s, the est^t_i component ratio is at least $\left(\frac{q}{w_{M^s_i}}\right)^{w_{M^s_i}/q}$.*

Proof We prove by induction on the steps taken from t. We omit the i superscript for M^i hereinafter.

At time t the only node with an est^t_i component is i with a ratio of one, 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, the invariant continues to hold.

1. Send: No effect on the invariant. The ratio at the sender stays the same, and w_M is unchanged.

2. Receive from $j \notin M_i^{s-1}$ by $k \notin M_i^{s-1}$: No effect on the invariant since no nodes in M are concerned.
3. Receive from $j \in M_i^{s-1}$ by $k \notin M_i^{s-1}$: Two things change: (1) $w_{M_i^s} = w_{M_i^{s-1}} + w_k^{s-1}$ and (2) k becomes a part of M_i. 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, $\alpha \geq \left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q}$. 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_i, is at least

$$\alpha \cdot \frac{q}{w_k^{s-1}} \overset{\text{induction hypothesis}}{\geq}$$

$$\geq \left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q} \frac{q}{w_k^{s-1}} \overset{\frac{w_k^{s-1}}{q} > 1}{>}$$

$$> \left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q} \left(\frac{q}{w_k^{s-1}}\right)^{w_k^{s-1}/q} >$$

$$> \frac{q^{\frac{w_{M_i^{s-1}}+w_k^{s-1}}{q}}}{\left(w_{M_i^{s-1}} + w_k^{s-1}\right)^{\frac{w_{M_i^{s-1}}}{q}} \left(w_{M_i^{s-1}} + w_k^{s-1}\right)^{\frac{w_k^{s-1}}{q}}} =$$

$$= \left(\frac{q}{w_{M_i^{s-1}} + w_k^{s-1}}\right)^{\frac{w_{M_i^{s-1}}+w_k^{s-1}}{q}} =$$

$$= \left(\frac{q}{w_{M_i^s}}\right)^{w_{M_i^s}/q} .$$

We conclude that the ratio at all the nodes in M_s is larger than the bound at s.
4. Receive from $j \notin M_i^{s-1}$ by $k \in M_i^{s-1}$: Denote the weight of the message by w_{in}. Two things change: (1) $w_{M_i^s} = w_{M_i^{s-1}} + w_{\text{in}}$ and (2) the ratio at k. The change of w_{M_i} 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 $\left(\frac{q}{w_{M_i^{s-1}}}\right)^{w_{M_i^{s-1}}/q}$. Therefore, after receiving the message, it is at least

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

We conclude that the ratio at all the nodes in M_s is larger than the bound at s.

5. Receive from $j \in M_i^{s-1}$ by $k \in M_i^{s-1}$: The ratio does not decrease below the minimum between the ratios in j and k, therefore the invariant's correctness follows directly from the induction hypothesis.

Lemma 8 *For any time $t > \overline{GST}$ and node i, there exists a time $t' > t$ after which every node j has an est_i^t component with ratio larger than $\left(\frac{q}{n}\right)^{n/q}$.*

Proof 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 message to all of its neighbors (lemma 6). Therefore, due to the connectivity of the network after $\overline{\text{GST}}$, and according to Lemma 7, eventually every node has an est_i^t component. Then we have $M_{\overline{\text{GST}}} = \mathcal{N}$, so $w_{M_{\overline{\text{GST}}}} = n$, and the ratio is not smaller than $\left(\frac{q}{n}\right)^{n/q}$.

5.2.4 Convergence

Theorem 1 (Liveness) *After $\overline{GST}$, the estimate at all nodes converges to the read average.*

Proof We construct a series of times $t_0, t_1, t_2, \ldots$ recursively, where the initial time is $t_0 = \overline{\text{GST}}$. For every t_{p-1} we define t_p to be a time from which each node $k \in \mathcal{N}^{\overline{\text{GST}}}$ has an est_i^{p-1} component with ratio at least $\left(\frac{q}{n}\right)^{n/q}$ for each $i \in \mathcal{N}^{\overline{\text{GST}}}$. Such a t_p exists according to Lemma 8.

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 8). 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 of 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 some value x.

Now, the values in *sent*, *received* and *cleared* are occasionally reset to the *est* value of their node (*sent*) or the neighbor's (*received* and *cleared*), and since we are after $\overline{\text{GST}}$, the weight *unrecvWeight* is zero. Since these all converge to x, their weighted sum E converges to the same value, i.e., $x = E$, and we have already shown (Lemma 1) that E is equal to the value of the read sum. We conclude that *est*, converges to the average read value.

We note that while *est* variables converge, the weights in the w variables fluctuate continuously, as nodes lose half their weight on send and similarly receive considerable weights.

5.2.5 Bounded State Variables

To conclude, we show that if the rate of dynamism allows the algorithm to converge between events, all state variables maintained by the nodes do not grow unboundedly.

Theorem 2 (Bounded variables) *If* `change` *and* `removeNeighbor` *events occur only when all* est_i *variables are in a* 2Δ *neighborhood of* R*, then all state variables are bounded.*

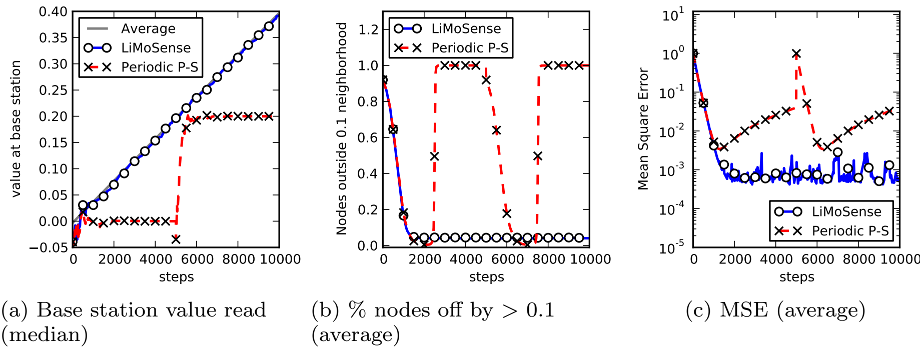
Proof Consider first the value component of the aggregates. A `change` event may significantly change the estimate value at a node if holds a small weight. Denote the maximal read value change by d_{change}, the maximal read value by M and the minimal read value by m. Since the minimal weight at a node is bounded by q, the change in bounded by d_{change}/q, and the estimate value is bounded in the range $[m - \Delta/q, M + \Delta/q]$. Since the variables $received_i(j)$, $sent_i(j)$, $totalDiff_i(j)$, and $unrecvVal_i$ are aggregates of *est*'s, their values are also bounded in the same range.

As for weights, the weight in $totalDiff_i(j)$ is explicitly bounded by having the sending node stop sending if its weight is too negative, and hence it is too positive on the opposite side (line 21). The weight of $received_i(j)$ is also explicitly bounded by resetting and changing epoch if its weight is too high (line 35). Bounding $received_i(j)$ automatically bounds $cleared_i(j)$, and $sent_j(i)$ is bounded explicitly (line 21). Finally, we bound the weight of est_i. Initially, the sum of weights in all *est* variables is n, and this sum changes on weight send/receive and on `removeNeighbor`. Send and receive cannot increase the sum of weights, since weight received was previously sent. On the other hand, `removeNeighbor` events change est_i by subtracting $totalDiff_i(j)$, possibly increasing/decreasing the sum of weights. Since the *totalDiff* weights are bounded by $2 \times$ `bound`, and each link could (temporarily) increment the sum of *est* weights in the system (if one side increases its estimate, and the other postpones the decrease to avoid negative weight), we conclude that the sum of *est* weights, and hence each *est* weight, is bounded by $n + 2 \times$ `bound` $\times n^2$. The *unrecvWeight* variables are similarly bounded, as they increase only on link failure by the difference of weights transferred on that link.

Note 1 If we replace the arbitrary neighbor choice of line 16 with a random choice from a static distribution, and take $q = 0$, then after $\overline{\text{GST}}$, the algorithm operates like classical push-sum. Therefore, in this case the estimates converge in an exponential rate [14, 5]. In this case the size of the values cannot be bounded, as the weights can be infinitesimally small.

6 Evaluation

Our formal analysis above shows that LiMoSense converges to the correct average during long enough periods of quiescence. In order to evaluate the behavior of LiMoSense during dynamic periods, we have conducted simulations of several representative scenarios. Our goal is to asses how fast the algorithm reacts to changes and succeeds to provide accurate information.



(a) Base station value read (median)

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

(c) MSE (average)

Fig. 1 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.

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).

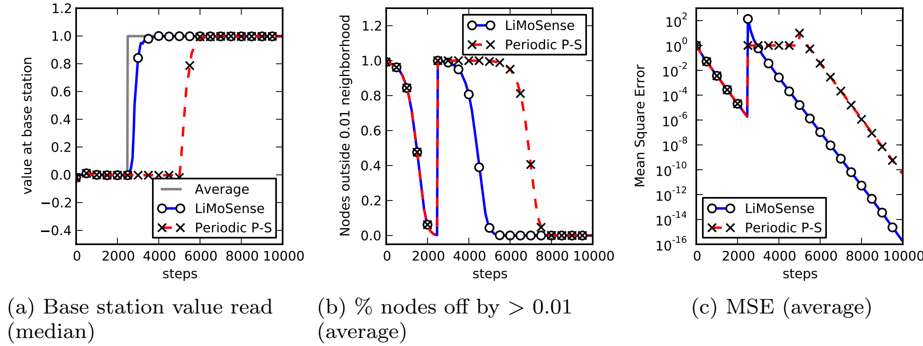
6.1 Methodology

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 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 sends a message. 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 algorithm's 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.



(a) Base station value read (median)

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

(c) MSE (average)

Fig. 2 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.

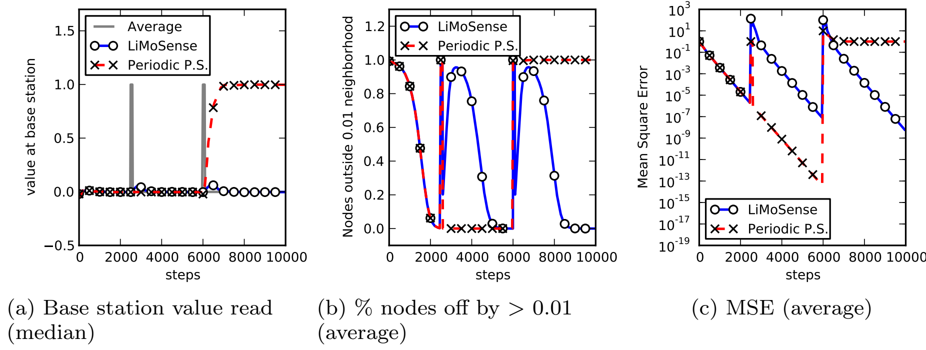
6.2 Slow monotonic increase

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, 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 1.

In Figure 1a 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, unable to follow the read average. The time it takes for convergence is so long that it never gets close the read average line.

In Figure 1b we see that after its initial convergence, the LiMoSense algorithm has most of the nodes maintain a good estimate of the read average with less than 10% of the nodes outside the 0.1 neighborhood. The restarting Push-Sum algorithm, on the other hand, has no nodes in this neighborhood most of the time, and most of the nodes in the neighborhood only for short intervals.

Finally, in Figure 1c 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 > 0.01 (average)

(c) MSE (average)

Fig. 3 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.

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 2.

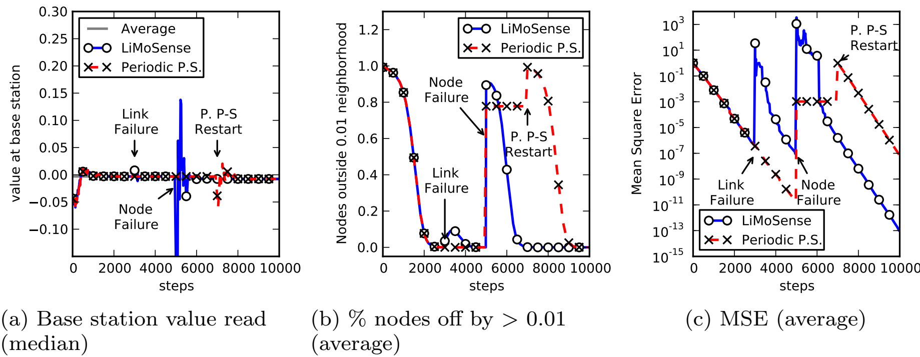
Figure 2a shows how the LiMoSense algorithm updates immediately after the shift, whereas the periodic Push-Sum algorithm updates at its first restart only. Figure 2b shows the ratio of erroneous sensors with error larger than 0.01 quickly dropping — right after the read average change for LiMoSense, and at restart for the periodic Push-Sum. Figure 2c shows the MSE decrease. 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 steps they return to their values before the shift. The initial values are taken from the standard normal distribution. The results are shown in Figure 3.

The LiMoSense algorithm's reaction is independent of 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



(a) Base station value read (median)

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

(c) MSE (average)

Fig. 4 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.

canceled, this value decreases. The estimate with respect to the read average is shown in Figure 3a, and the ratio of correct sensors is in Figure 3b. The impulse essentially restarts the MSE convergence, as shown in Figure 3c — 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 3a–3c 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 3a 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, about 7 links fail, and respective nodes employ their `removeNeighbor` functions. We see the effect of this link removal in Figure 4. In Figure 4a the effect can hardly be seen, but a temporary decrease of the accurate nodes can be seen in Figure 4b, and in Figure 4c 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 of its neighbors call their `removeNeighbor` functions. Figure 4a shows the extreme noise at the base station caused by the failure, and in Figure 4b

we see the ratio of inaccurate nodes rising sharply before converging again. We see in Figure 4c that the node removal effectively requires 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 4a.

7 Conclusion

We have 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 dynamically tracks and aggregates a large collection of ever-changing sensor reads. It overcomes message loss, node failures and recoveries, and dynamic network topology changes. The main focus of this work has been the formal analysis of LiMoSense's correctness, namely showing it converges to the read average once the system stabilizes. For completeness, we have also demonstrated the behavior of LiMoSense in representative dynamic scenarios, showing its fast convergence rate.

acknowledgements The authors thank an anonymous reviewer for important comments on an earlier version of this work.

References

1. Paulo Sérgio Almeida, Carlos Baquero, Martin Farach-Colton, Paulo Jesus, and Miguel A. Mosteiro. Fault-tolerant aggregation: Flow updating meets mass distribution. In *OPODIS*, 2011.
2. G. Asada, M. Dong, T.S. Lin, F. Newberg, G. Pottie, W.J. Kaiser, and H.O. Marcy. Wireless integrated network sensors: Low power systems on a chip. In *ESSCIRC*, 1998.
3. Yitzhak Birk, Idit Keidar, Liran Liss, and Assaf Schuster. Efficient dynamic aggregation. In *DISC*, 2006.
4. Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Gossip algorithms: design, analysis and applications. In *INFOCOM*, 2005.
5. Stephen P. Boyd, Arpita Ghosh, Balaji Prabhakar, and Devavrat Shah. Randomized gossip algorithms. *IEEE Transactions on Information Theory*, 52(6):2508–2530, 2006.
6. J-Y Chen, Gopal Pandurangan, and Dongyan Xu. Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 17(9):987–1000, 2006.
7. Ittay Eyal, Idit Keidar, and Raphael Rom. LiMoSense – live monitoring in dynamic sensor networks. In *7th International Symposium on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSOR'11)*, 2011.
8. Fabio Fagnani and Sandro Zampieri. Randomized consensus algorithms over large scale networks. *IEEE Journal on Selected Areas in Communications*, 26(4):634–649, 2008.
9. Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2), 1985.
10. Navendu Jain, Prince Mahajan, Dmitry Kit, Praveen Yalagandula, Michael Dahlin, and Yin Zhang. Network imprecision: A new consistency metric for scalable monitoring. In *OSDI*, 2008.
11. Márk Jelasity and Alberto Montresor. Epidemic-style proactive aggregation in large overlay networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 102–109. IEEE, 2004.
12. Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3), 2005.
13. Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Fault-tolerant aggregation for dynamic networks. In *SRDS*, 2010.
14. David Kempe, Alin Dobra, and Johannes Gehrke. Gossip-based computation of aggregate information. In *FOCS*, 2003.
15. Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.
16. Damon Mosk-Aoyama and Devavrat Shah. Computing separable functions via gossip. In *PODC*, 2006.
17. Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, 2004.
18. Andrew S. Tanenbaum. *Computer networks*. Prentice Hall, 2003.
19. Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: communicating with a cubic-millimeter computer. *Computer*, 34(1), 2001.
20. Fetahi Wuhib, Mads Dam, Rolf Stadler, and Alexander Clem. Robust monitoring of network-wide aggregates through gossiping. *IEEE Transactions on Network and Service Management*, 6(2):95–109, 2009.