How to Implement Doubly-Stochastic Matrices for Consensus-Based Distributed Algorithms

Sergio Valcarcel Macua , Carlos Moreno Leon , Jhoan Samuel Romero , Silvana Silva Pereira
Javier Zazo , Alba Pagès-Zamora , Roberto López-Valcarce and Santiago Zazo

***Abstract*—Doubly-stochastic matrices are usually required by consensus-based distributed algorithms. We propose a simple and efficient protocol and present some guidelines for implementing doubly-stochastic combination matrices even in noisy, asynchronous and changing topology scenarios. The proposed ideas are validated with the deployment of a wireless sensor network, in which nodes run a distributed algorithm for robust estimation in the presence of nodes with faulty sensors.**

***Index Terms*—Consensus, Contiki OS, distributed algorithms, robust estimation, wireless sensor network.**

I. Introduction

The advantages of distributed learning algorithms over networks with respect to centralized schemes have long been recognized (see, e.g., the surveys [1], [2]). Consensus strategies are widely used for implementing distributed algorithms and have recently found many applications (e.g., feature extraction, adaptive filtering, classification, clustering, detection, estimation, and convex optimization methods, among others). Consensus-based distributed algorithms usually require a doubly-stochastic matrix for performing the combination of the information flowing across the network. However, although there are several mathematical methods to design such matrices, in practice, when communications or synchronization among nodes are not perfect, even theoretically well designed combination matrices become nondoubly-stochastic, resulting into biased results.

There are few implementations of distributed algorithms [3]–[6]. However, they are sensitive to asynchronous and noisy communications (e.g., in wireless-sensor-networks). This paper proposes a simple protocol for guaranteeing that the combination matrix is doubly-stochastic at every iteration, independently of the amount of packet loss and the lack of synchronization among nodes. The protocol is also robust against permanent changes in the topology (e.g., because of node failure or after adding new nodes). We also provide some implementation guidelines to abstract the implementation from the mathematical formulation of the algorithms, so the designer can use consensus-like combinations as a service and focus on the data-processing step. The proposed methods are validated experimentally with the implementation of a sophisticated robust estimation algorithm introduced in [7].

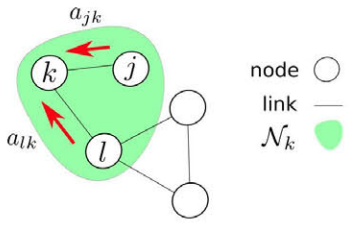

Fig. 1. Example of networks considered in this work.

II. Consensus Algorithms

Consider a network of N cooperative agents with arbitrary topology. The network is modeled by a graph where the nodes are the agents, and edges represent the communication links (see Figure 1). We assume the graph is connected (i.e., there is at least one path between any pair of nodes). The agents want to estimate some parameter vector, w^o (e.g., the sample estimate of a sufficient statistic of the data, or the common minimizer of some objective function). If each individual agent has only access to a subset of the data, its individual learning process will be biased. Nevertheless, by cooperating with its neighbors each node can approach the same performance as the one of a centralized architecture.

Let $\mathcal{N}_k$ denote the neighborhood of node k (i.e., all the nodes that can share information with node k, including k itself). Let a_{lk} denote the non-negative weight given by node k to the information shared by node l. Many consensus algorithms can be expressed by the following two steps [8]:

$$\phi_{k,i-1} = \sum_{l \in \mathcal{N}_k} a_{lk} w_{l,i-1} \tag{1a}$$

$$w_{k,i} = \phi_{k,i-1} - \mu_k s_{k,i}(w_{k,i-1}) \tag{1b}$$

where $w_{k,i}$ denotes the estimate of the parameter of interest at time i by node k, $\phi_{k,i}$ is an intermediate variable, and $s_{k,i}(\cdot)$ is some local function that depends on the problem at hand (e.g., the gradient of a local objective function in a distributed optimization problem), and μ_k is the step-size of the update. In (1a), each node combines the estimates of its neighbors (including itself) with the corresponding weights. Then, in (1b), each node processes information locally.

We collect the weights a_{lk} into a combination matrix A of size $N \times N$. In consensus algorithms, A is designed to satisfy the following three conditions:

$$\rho\left(A^\top - \frac{1}{N}\mathbb{1}\mathbb{1}^T\right) < 1, \quad A^\top \mathbb{1} = \mathbb{1}, \quad A\mathbb{1} = \mathbb{1} \qquad (2)$$

where $\rho(\cdot)$ denotes the spectral radius. The first condition ensures asymptotic convergence, while the other two determine the convergence point and amount to saying that A is doubly stochastic.

As an illustrative example of the algorithm (1a)–(1b), consider a network of agents, in which each agent k wants to estimate the sample mean of the observations of all the agents across the network, but it only has access to its own observation x_k. This problem is solved by the standard average consensus rule (see, e.g., [2]), given by

$$\phi_{k,i-1} = \sum_{l \in \mathcal{N}_k} a_{lk} w_{l,i-1}, \quad w_{k,i} = \phi_{k,i-1} \qquad (3)$$

where the initial estimate is set equal to the local observation (i.e., $w_{k,0} = x_k$). For simplicity, assume observations are scalar valued, and introduce the vector $w_i = [w_{1,i}, \ldots, w_{N,i}]^\top$ with entries the individual estimates of all the nodes. Then, we can express (3) as a network recursion: $w_i = A^\top w_{i-1}$. It is well known that when A satisfies (2), then $\lim_{i\to\infty} A^i = \frac{1}{N}\mathbb{1}\mathbb{1}^T$. Hence, every node will approach the network average

$$\lim_{i\to\infty} w_i = \lim_{i\to\infty} A^i w_0 = \frac{1}{N}\sum_{k=1}^{N} x_k \triangleq w^o \qquad (4)$$

III. The Wait-for-the-Slowest (WFS) protocol

When aiming to implement (1a)–(1b), the conditions in (2) become relevant. In particular, left-stochasticity (i.e., $A^\top \mathbb{1} = \mathbb{1}$) means that the weights given by each agent to all its neighbors add up to one. This condition is easy to enforce even when some packets are lost, because the node could recalculate the weights at every iteration giving positive weight only to the packets that were successfully received (including its own). On the other hand, ensuring right stochasticity (i.e., $A\mathbb{1} = \mathbb{1}$) is more challenging because each row depends on the whole neighborhood. It means that the weights given by the neighbors of some node to the information coming from that node must add up to one. In a real scenario (e.g., under wireless communications), this condition may be difficult to satisfy. Consider that some node broadcasts a packet, which is only received by a subset of its neighbors. Those neighbors that received the packet can give positive weight and compute (1a). However, those that did not receive the packet must set the corresponding weight to zero and, thus, the right stochastic condition will not hold. Therefore, the algorithm will converge to a *weighted* sample average, so that the result of (3) will be biased.

Besides dealing with lossy links, another issue is how to keep synchronization among neighbors, so they update its estimate at the same rate. Consider a simple network of just two nodes, k and l, which are performing periodic iterations of the form (1a)–(1b). Assume the clock of k has a drift so, eventually, it will be out of synch and, although it may receive data coming from node l, it will not transmit any packet. In this case, node k has all the required information to perform (1a) and, then, to update its estimate using (1b). On the other hand, node l is still waiting for the data coming from node k, so it can not update its estimate at that iteration. It means that, during the update of k, the combination matrix A is only left stochastic, as opposed to doubly stochastic, and, again, the asymptotic result will be biased.

TABLE I
Neighbor table for node k of Figure 1.

$\mathcal{N}_\cdot$	$\lvert\mathcal{N}_\cdot\rvert$	$a_{\cdot k}$ (using Metropolis rule [1], [2])
j	2	$a_{jk} = 1/2$
l	4	$a_{lk} = 1/4$
k	3	$a_{kk} = 1 - (1/2 + 1/4) = 1/4$

A. Basic design for tackling temporary link failures

In order to tolerate packet loss, asynchronous transmissions and other impairments of real environments, and avoid biased results of the distributed algorithm, some sort of coordination among neighbors becomes necessary, so they can give total unit weight to node k at every iteration (thus, making the combination matrix row stochastic). We propose an efficient and easy to implement mechanism, named Wait-For-the-Slowest (WFS) protocol. The idea underlying WFS is simple: synchronization at the application layer (rather than at the MAC level). In particular, every node has a table where it stores an iteration counter for each of its neighbors, so it waits until this table is filled with the correct information before performing the combination step. This mechanism works efficiently under the assumptions that every node knows its neighbors before performing the consensus iterations. Indeed, WFS is composed of two main stages: an initial *setup* and the *consensus* loop.

The *setup* stage is executed only once, at the beginning of the algorithm, with the purpose of building the combination matrix. During this initial stage, every node broadcasts 'discovery' messages with some useful local information (e.g., $|\mathcal{N}_k|$ for the Metropolis rule [1], [2]). Let us say that node l receives a discovery message from node k, then, it checks whether node k was already in its neighbor list; if it was not, another row is added to the table; otherwise, just the degree $|\mathcal{N}_\cdot|$ and the weight $a_{\cdot k}$ are updated (see Table I for an example of neighbor table). At the end of the setup stage, every node has stored its neighbor table, where the weight column can be thought of as a column of the combination matrix.

Once the nodes know their neighborhoods and have set the corresponding combination weights, they start the *consensus* stage. When some event (which could be asynchronous, such as the arrival of a new packet or the availability of a new sample, or periodic, in synch with an internal iteration timer) triggers the combination step (1a), the node has to be sure that it has the right estimates. Let us say that node k wants to update $w_{k,i}$, then it should combine $w_{l,i-1}$ for all $l \in \mathcal{N}_k$, instead of any other $w_{l,j}$ with $j \neq i-1$. If such information

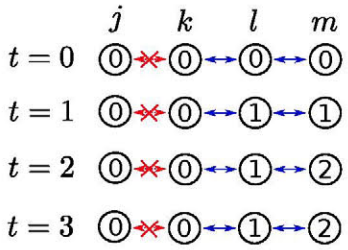

Fig. 2. Example 1. A chain of nodes running WFS, where the link between the two nodes in the left end of the chain is faulty. The iteration number (i) of the estimate (e.g., $w_{k,i}$) is written inside each circle. At time $t = 1$, node j cannot update because info from node k is missing. Similarly, although node k has info from node l, it misses the estimate from node j, so it cannot update its estimate. On the other hand, nodes l and m have all the information from their neighborhoods. Nevertheless, at time $t = 2$, node l misses $w_{k,1}$ so it has to stop. Node m have both $w_{l,1}$ and $w_{m,1}$ so it can compute $w_{m,2}$. At time $t = 3$, node m is missing $w_{l,2}$ so it is forced to stop too and, hence, the whole network has to either wait until the link between j and k works again (some retransmission finally reaches the node) or until a retransmission timeout, which they understand as a permanent topology change so they safely remove each other from their neighbor tables.

is not available, the node patiently waits for a retransmission of the missing information. Note that when one node stops updating its estimate (because it misses information from any of its neighbors), then the rest of its neighbors will also wait for it. Therefore, the difference in the iteration number between two neighbors is at most 1 (e.g., $w_{k,i}$ and $w_{l,i-1}$). Thus, at every iteration, every node has to transmit two parameters, namely, the two most recent estimates (i.e., $w_{k,i}$ and $w_{k,i-1}$). This is illustrated in Figure 2.

B. Extension for dealing with permanent topology changes

Although the mechanism explained above is robust against temporary link failures (e.g., due to fading, collisions...) and asynchronous communications (e.g., because of clock drifts, different transmission rates...), two extensions are required in order to tackle permanent changes in the topology: a *link-timer* that measures how long a link has been inactive, and a *dynamic neighbor table* that can be updated during the consensus stage.

When one node receives a packet from one of its neighbors, it resets the link-timer. If one link fails permanently (e.g., due to node failure), the timer of each of the affected nodes will reach a tunable timeout period, hence, they will assume that the link will not recover anymore (at least in the short term) and remove it from their tables. The iteration number is a form of acknowledgement that the information is diffusing properly. Therefore, every node keeps transmitting its relevant parameters as described in Sec. III-A. Note that if the link-failures divide the network in two or more islands, then WFS will still work in each of the islands, but they may independently converge to different results.

When a new node is added to the network after the network has entered the consensus stage, the other nodes will not take it into account (they will continue giving zero weight). Nevertheless, if the nodes are able to update their neighbor tables every time they receive a packet from an unknown neighbor, even during the consensus stage, then the network

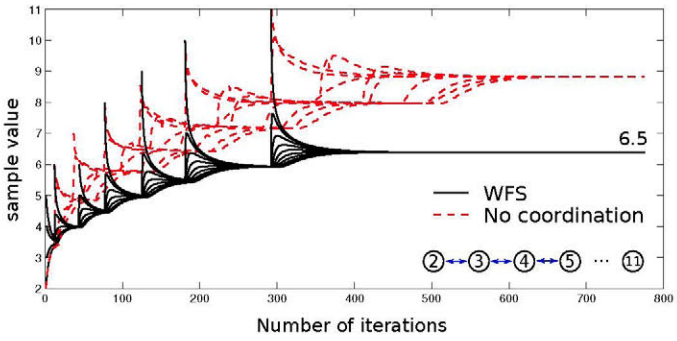

Fig. 3. Robustness of WFS. We begin with a chain of 4 nodes, with values $\{2, 3, 4, 5\}$, respectively. Then, we add up to 6 more nodes, one at a time, with values $\{6, 7, 8, 9, 10, 11\}$, respectively. Note that WFS is able to achieve the sample average of all the available nodes at each time (resulting in 6.5 when all nodes are included in this case).

will be able to integrate new nodes on the fly. When the new node boots, it enters the setup-stage, in which it broadcasts discovery messages and listens to discover its neighbors. During this stage, apart of storing its neighbors' identity, it also keeps their iteration number. At the end of this stage, the node sets its iteration number equal to the highest iteration number among its neighbors. This way, the new node will be included in new iterations of the consensus algorithm without biasing earlier uncompleted iterations in its neighborhood.

Fig. 3 shows the robustness of WFS against noisy links, asynchronous updates and permanent changes in the topology.

IV. Implementation Guidelines

The nodes will operate in an asynchronous environment, hence, we find convenient to use an event-oriented operating system with multi-threading capabilities.

We define a 2-thread template to split the communication-tasks performed by each node: processing asynchronous incoming packets (RX) and synchronously transmitting its own data (TX) (other tasks, like sensing/actuating, could be included into the RX or TX threads or into another thread at convenience). The state-logic of the WFS protocol is embedded in the RX thread, which triggers the adaptation step and checks whether the updated estimates of all the neighbors are available before triggering the combination step. Therefore, the RX thread controls the update of the two most recent estimates, which will be broadcast by the TX thread. This implementation scheme provides reliable data-diffusion at every consensus iteration (i.e., it ensures a doubly stochastic matrix during the combination step (1a)), Therefore, the developer can abstract from the within-neighborhood communications and focus on testing the local adaptation step (1b), speeding up the development process. Still, a complete and flexible simulator, able to emulate communication impairments, changes in the topology and asynchronous events is a desirable feature. A further constraint when developing distributed algorithms for large scale networks is that the hardware of the nodes often has little available memory. For these reasons, we choose Contiki

OS, which brings a powerful simulator, named Cooja, and offers multithreading functionality plus a complete communication stack—we use UDP over IPv6—at a minimum memory footprint (in the order of tens of kilobytes), and it is designed to be programmed with standard C language (which makes debugging simpler as opposed to other options, like TinyOS).

V. Case of study: distributed robust estimation

In this section, we illustrate the effectiveness of WFS as a solution for implementing sophisticated distributed algorithms. Consider a network of N sensors collecting one observation each under the following model

$$y_k = b_k w^o + n_k, \qquad k = 1, \ldots, N, \tag{5}$$

where w^o is the parameter of interest, $\{b_k, \forall k\} = \{0, 1\}$ are i.i.d. Bernoulli random variables with probability $p \triangleq \Pr\{b_k = 1\}$, and $\{n_k, \forall k\}$ are i.i.d. zero-mean Gaussian with variance σ^2 and independent of $\{b_k, \forall k\}$. The nodes run a distributed robust algorithm named DB-DEM [7], which provides an unbiased estimation even under the presence of nodes with faulty sensors that report only noise (i.e., those for which $b_k = 0$). Let $w_{k,i}$ and $\sigma_k^2(i)$ denote the estimates of the parameters at node k at time i, and $p_k(i)$ the a posteriori probability of b_k given $\{y_k, w_{k,i}, \sigma_k^2(i)\}$.

The DB-DEM algorithm diffuses $p_k(i)$ across the network by means of local combination steps of four intermediate variables (one combination step each), which are then used to update the local estimates. In this way, an initial period for information diffusion is gradually switched off at the same time as an averaging process is gradually switched on.

The followed methodology consists in: 1) Developing the algorithm in a numerical computations environment (e.g., MATLAB) and defining a set of test-vectors (input-output) that guarantee the correct operation of the algorithm; 2) Developing the same program in C language, splitting the adaptation step (local computations) and the combination step (within neighborhood communications), but still assuming that the combination step is performed exactly; 3) Moving the local computations to the Contiki OS multi-threading template—by using WFS as the service that provides the combination step—and checking the test-vectors in the Cooja simulator. 4) Customizing the Cooja code for the hardware platform and checking the test-vectors in the real deployment. 5) Developing sensing-acting routines for the hardware platform.

A chain of nodes and a fully connected network are the least favorable topologies for testing: in the former the information takes several steps to diffuse across the network; in the latter, all the nodes interfere each other and many packets are lost. Figure 4 shows the results for robust estimation vs. average consensus for a simulated chain of 10 nodes. We compare perfect communications (Matlab) vs. the WFS implementation in Cooja under 30% of packet loss and nodes with different transmission rates. Both simulations match almost perfectly.

In addition to simulations, we ported the DB-DEM algorithm to real hardware with minimal effort (note that Cooja already simulated the operating system, with the exception of the sensing and actuating routines). We used commercial off-the-shelf Econotag motes, equipped with MC13224 System-on-Chip, which offers an ARM7 microcontroller at 24MHz with 96 KB RAM and an integrated 802.15.4 radio. The results obtained with the real motes match closely the simulations in Cooja and Matlab. Moreover, WFS performed efficiently, being able to complete one iteration per second in different topologies, including a fully connected network of 10 nodes. A complete description of the experiments with real motes is left for a future extended version of this work.

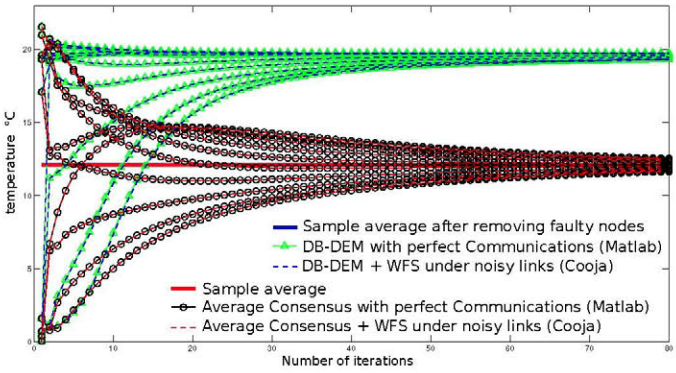

Fig. 4. DB-DEM vs sample average, with perfect communications vs. WFS and noisy and asynchronous links. The simulated network consists of 10 nodes in chain topology, where 4 of them have a faulty sensor. DB-DEM is able to detect and compensate the faulty readings. Moreover, WFS (dashed) behaves similar to the curves that assume perfect communication (solid with marker).

VI. Conclusions

We introduced WFS: an efficient protocol that performs the combination step of the consensus algorithm even under asynchronous and impaired communications. By using WFS, the designer can focus on implementing the local adaptation step, speeding up the development cycle.

References

[1] A. H. Sayed, "Diffusion adaptation over networks," in Academic Press Library in Signal Processing, R. Chellapa and S. Theodoridis, Eds. Elsevier, 2014, vol. 3, pp. 323–454. Also available as arXiv:1205.4220v1, May 2012.

[2] F. Garin and L. Schenato, "A Survey on Distributed Estimation and Control Applications Using Linear Consensus Algorithms," in *Networked Control Systems*. Springer, 2011, vol. 406, pp. 75–107.

[3] W. Ren, H. Chao, W. Bourgeous, N. Sorensen, and Y.-Q. Chen, "Experimental implementation and validation of consensus algorithms on a mobile actuator and sensor network platform," in *IEEE Int. Conf. on Systems, Man and Cybernetics*, Oct 2007, pp. 171–176.

[4] R. Pagliari and A. Scaglione, "Implementation of average consensus protocols for commercial sensor networks platforms," in *Grid Enabled Remote Instrumentation*. Springer, 2009, pp. 81–95.

[5] J. Kenyeres, M. Kenyeres, and M. Rupp, "Experimental node failure analysis in WSNs," in *Int. Conf. on Systems, Signals and Image Processing*, June 2011, pp. 1–5.

[6] J. Kenyeres, M. Kenyeres, M. Rupp, and P. Farkas, "Wsn implementation of the average consensus algorithm," in *European Wireless*, April 2011, pp. 1–8.

[7] S. Pereira, R. Lopez-Valcarce, and A. Pages-Zamora, "A diffusion-based em algorithm for distributed estimation in unreliable sensor networks," *Signal Processing Letters, IEEE*, vol. 20, no. 6, pp. 595–598, June 2013.

[8] J. Chen and A. H. Sayed, "On the learning behavior of adaptive networks — Part I: Transient analysis," *submitted for publication* [also available as arXiv:1312.7581], 2013.