Universidade do Minho
Escola de Engenharia

Paulo César de Oliveira Jesus

Robust Distributed Data Aggregation

Setembro de 2011

Paulo César de Oliveira Jesus **Robust Distributed Data Aggregation**

UMinho | 2011

Universidade do Minho
Escola de Engenharia

Paulo César de Oliveira Jesus

Robust Distributed Data Aggregation

Tese de Doutoramento
Programa Doutoral em Informática MAP-i

Trabalho efectuado sob a orientação do
Doutor Carlos Miguel Ferraz Baquero Moreno
e do
Doutor Paulo Sérgio Soares de Almeida

Setembro de 2011

Acknowledgments

I would like to express my gratitude to all the persons that helped and supported me along my PhD. I dedicate this work to all of them.

In particular, I would like to thank:

- my supervisors, Prof. Carlos Baquero and Prof. Paulo Sérgio Almeida for their valuable guidance, and all the advices, comments and suggestions provided throughout this study;
- Telma for her love, constant support and motivation;
- my colleagues from the lab, for their companionship, team spirit and joy;

Finally, I would like to acknowledge the committee of the MAP-i Doctoral Program and FCT (Fundação para a Ciência e Tecnologia) for the provided PhD grant (SFRH/BD/33232/2007), which allowed me to dedicate these last few years to this research work.

Robust Distributed Data Aggregation

Distributed aggregation algorithms are an important building block of modern large scale systems, as it allows the determination of meaningful system-wide properties (e.g., network size, total storage capacity, average load, or majorities) which are required to direct the execution of distributed applications. In the last decade, several algorithms have been proposed to address the distributed computation of aggregation functions (e.g., COUNT, SUM, AVERAGE, and MAX/MIN), exhibiting different properties in terms of accuracy, speed and communication tradeoffs. However, existing approaches exhibit many issues when challenged in faulty and dynamic environments, lacking in terms of fault-tolerance and support to churn.

This study details a novel distributed aggregation approach, named *Flow Updating*, which is fault-tolerant and able to operate on dynamics networks. The algorithm is based on manipulating flows (inspired by the concept from graph theory), that are updated using idempotent messages, providing it with unique robustness capabilities. Experimental results showed that Flow Updating outperforms previous averaging algorithms in terms of time and message complexity, and unlike them it self adapts to churn and changes of the initial input values without requiring any periodic restart, supporting node crashes and high levels of message loss.

In addition to this main contribution, others can also be found in this research work, namely: a definition of the aggregation problem is proposed; existing distributed aggregation algorithm are surveyed and classified into a comprehensive taxonomy; a novel algorithm is introduced, based on Flow Updating, to estimate the Cumulative Distribution Function (CDF) of a global system attribute.

It is expected that this work will constitute a relevant contribution to the area of distributed computing, in particular to the robust distributed computation of aggregation functions in dynamic networks.

Agregação Distribuída e Robusta

Os algoritmos de agregação distribuídos têm um papel importante no desenho dos sistemas de larga escala modernos, uma vez que permitem determinar o valor de propriedades globais do sistema (e.g., tamanho da rede, capacidade total de armazenamento, carga média, ou maiorias) que são fundamentais para a execução de outras aplicações distribuídas. Ao longo da última década, diversos algoritmos têm sido propostos para calcular funções de agregação (e.g., CONTAGEM, SOMA, MÉDIA, ou MIN/MAX), possuindo diferentes características em termos de precisão, velocidade e comunicação. No entanto, as técnicas existentes exibem vários problemas quando executadas em ambientes com faltas e dinâmicos, deixando a desejar em termos de tolerância a faltas e não suportando a entrada/saída de nós.

Este estudo descreve detalhadamente uma nova abordagem para calcular funções de agregação de forma distribuída, denominada *Flow Updating*, que é tolerante a faltas e capaz de operar em redes dinâmicas. O algoritmo é baseada na manipulação de fluxos (inspirado no conceito da teoria de grafos), que são atualizados por mensagens idempotentes, conferindo-lhe capacidades únicas em termos de robustez. Os resultados experimentais demonstram que o *Flow Updating* supera os anteriores algoritmos baseados em técnicas de *averaging* em termos de complexidade de tempo e mensagens, e, ao contrário destes, auto adapta-se a mudanças da rede (i.e., entrada/saída de nós e alteração dos valores iniciais) sem necessitar de reiniciar periodicamente a sua execução, suportando falhas de nós e elevados níveis de perdas de mensagens.

Para além desta contribuição principal, outras são também encontradas neste trabalho, nomeadamente: é proposta uma definição do problema da agregação; é descrito o estado da arte em termos dos algoritmos de agregação distribuídos, e estes são classificados numa taxonomia abrangente; é apresentado um novo algoritmo baseado no *Flow Updating* para estimar a Função de Distribuição Cumulativa (CDF) de um atributo global do sistema.

Espera-se que este trabalho constitua um contributo relevante para a área da computação distribuída, em especial para a computação distribuída e robusta de funções de agregação em redes dinâmicas.

Contents

I Exordium

The Purpose of this Work **1**

1 Introduction **3**

1.1 Motivation 4

1.2 Contributions 6

1.3 Organization 7

2 Problem Definition **9**

2.1 Properties of Aggregation Functions 10

2.1.1 Decomposability 10

2.1.2 Duplicate sensitiveness and idempotence 13

2.2 Taxonomy of common aggregation functions 14

II State of the Art **15**

3 Related Work **17**

3.1 Communication 18

3.1.1 Hierarchy-based 20

3.1.2 Ring 27

3.1.3 Flooding/Broadcast 28

3.1.4 Random Walk 29

3.1.5 Gossip-based 30

3.1.6 Hybrid 33

3.2 Computation 36

3.2.1 Hierarchical 37

3.2.2 Averaging 38

3.2.3 Sketches . . . 42
3.2.4 Digests . . . 47
3.2.5 Counting . . . 51

4 Dependability Issues of Existing Algorithms 57
4.1 Robustness of Averaging Algorithms . . . 59
4.1.1 Push-Sum Protocol . . . 59
4.1.2 Push-Pull Gossiping . . . 60
4.1.3 Distributed Random Grouping . . . 62

III Robust Distributed Aggregation Approach 65

5 Flow Updating 67
5.1 Algorithm . . . 69
5.2 Correctness . . . 72
5.2.1 Model and Assumptions . . . 73
5.2.2 (Simplest) Non Concurrent Model . . . 74
5.2.2.1 Message Loss . . . 81
5.2.3 Concurrent Model (with non overlapping groups) . . . 94
5.2.3.1 Message Loss . . . 101
5.3 Variations and Improvements . . . 102
5.3.1 Flow Updating with Preferential Grouping . . . 102
5.3.1.1 Formation of Averaging Groups . . . 105
5.3.2 Termination/Quiescence . . . 107
5.3.3 Asynchrony . . . 108

6 Estimating Complex Aggregates 111

IV Evaluation 117

7 Evaluation 119
7.1 Simulation Settings . . . 120
7.2 Comparison Against Other Algorithms . . . 121
7.3 Fault-Tolerance . . . 128
7.4 Flow Updating with Preferential Grouping . . . 131

7.5 Dynamism . . . 134
7.5.1 Churn . . . 134
7.5.1.1 Fault Detection . . . 138
7.5.2 Input Values Change . . . 142
7.6 Termination/Quiescence . . . 143
7.7 Asynchrony . . . 151

V Peroratio
Achievements and Future Work **155**

8 Conclusion **157**
8.1 Future Work . . . 159

VI Appendices **161**

A Modeling *Flow Updating* as a Difference Equation **163**
A.1 State Model . . . 164
A.2 Message Model . . . 166
A.3 Example . . . 168
A.3.1 Scenario 1 (tree network) . . . 169
A.3.2 Scenario 2 (multi-path network) . . . 171
A.4 Problem . . . 173
A.4.1 Additional information . . . 173
A.4.1.1 Exploration . . . 173
A.4.1.2 Similarity between models: State *vs* Message . . . 176
A.4.1.3 Properties of matrix A . . . 177

Bibliography **179**

List of Figures

2.1 Examples of computation of a self-decomposable aggregation function (associative and commutative). 12

4.1 Violation of the mass conservation invariant in the Push-Pull Gossiping protocol (estimates at the end of each round). 61

5.1 Example (non concurrent model) depicting the effect of message loss. 86
5.2 Example (non concurrent model) where the same node is chosen to execute the algorithm after message loss. 89
5.3 Example (non concurrent model) where a node that fails to receive a message from the previous round is chosen to execute the algorithm. . 90
5.4 Example (non concurrent model) where a node that successfully received a message from the previous round is chosen to execute the algorithm. 91
5.5 Example (non concurrent model) where a node that did not participate in the previous round is chosen to execute the algorithm. 92
5.6 Example (non concurrent model) where the node chosen to execute the algorithm belongs to a link where a message was previously lost. . . . 93

7.1 Comparison of *Flow Updating* against other averaging algorithms, on random networks with $d \approx 3$ and different sizes n (i.e., 100, 1000, 10000). 122
7.2 Comparison of *Flow Updating* against other averaging algorithms, on random networks with $d \approx 10$ and different sizes n (i.e., 100, 1000, 10000). 123
7.3 Comparison of *Flow Updating* against other averaging algorithms, on attach networks with $d \approx 3$ and different sizes n (i.e., 100, 1000, 10000). 124

7.4 Comparison of *Flow Updating* against other averaging algorithms, on attach networks with $d \approx 10$ and different sizes n (i.e., 100, 1000, 10000). 125
7.5 Comparison of *Flow Updating* against other averaging algorithms, on 2D/mesh networks with $d \approx 3$ and different sizes n (i.e., 100, 1000, 10000). 126
7.6 Comparison of *Flow Updating* against other averaging algorithms, on 2D/mesh networks with $d \approx 10$ and different sizes n (i.e., 100, 1000, 10000). 127
7.7 *Flow Updating* with message loss on random networks ($n = 1000$, $d \approx 3$). 129
7.8 *Flow Updating* with message loss on random networks ($n = 1000$, $d \approx 10$). 129
7.9 *Flow Updating* with message loss on 2D/mesh networks ($n = 1000$, $d \approx 3$). 130
7.10 *Flow Updating* with message loss on 2D/mesh networks ($n = 1000$, $d \approx 10$). 130
7.11 Comparison of *Flow Updating* against its variations, on random networks with size $n = 1000$ and different average connection degrees (i.e., 3 and 10). 132
7.12 Variations of *Flow Updating* with loss – random networks ($n = 1000$; $d \approx 3$). 132
7.13 Variations of *Flow Updating* with loss – random networks ($n = 1000$; $d \approx 10$). 132
7.14 Comparison of *Flow Updating* against its variations, on 2D/mesh networks with size $n = 1000$ and different average connection degrees (i.e., 3 and 10). 133
7.15 Variations of *Flow Updating* with loss – 2D/mesh networks ($n = 1000$; $d \approx 3$). 133
7.16 Variations of *Flow Updating* with loss – 2D/mesh networks ($n = 1000$; $d \approx 10$). 133
7.17 Comparison of FU in dynamic settings, with no message loss – random networks ($n = 1000$, $d \approx \log n$). 135

7.18 FU in dynamic settings, with message loss – random networks ($n = 1000$, $d \approx \log n$). 136
7.19 FU in dynamic settings, with message loss – 2D/mesh networks ($n = 1000$, $d \approx \log n$). 137
7.20 Estimates distribution of FU in dynamic settings, with 20% of message loss. 138
7.21 Effect of FD on the execution of FU in dynamic settings with message loss – random networks ($n = 1000$, $d \approx \log n$). 140
7.22 Mistakes of FD in dynamic settings with message loss – random networks ($n = 1000$, $d \approx \log n$). 141
7.23 FU with input value changes, and message loss – random networks ($n = 1000$, $d \approx 3$). 143
7.24 Quiescence with no message loss – random networks ($n = 1000$, $d \approx 3$). 145
7.25 Quiescence with 10% of message loss – random networks ($n = 1000$, $d \approx 3$). 146
7.26 Quiescence with 20% of message loss – random networks ($n = 1000$, $d \approx 3$). 147
7.27 Quiescence with no message loss – 2D/mesh networks ($n = 1000$, $d \approx 10$). 148
7.28 Quiescence with 10% of message loss – 2D/mesh networks ($n = 1000$, $d \approx 10$). 149
7.29 Use of different threshold values to leave quiescence – random networks ($n = 1000$, $d \approx 3$), with no message loss. 151
7.30 Execution of FU in asynchronous setting, using an average message transmission of 89.88 ms – random networks ($n = 1000$, $d \approx 3$). . . . 153

A.1 Tree network topology 169
A.2 Multi-path network topology 171

List of Algorithms

1 Flow Updating algorithm. 69
2 Simplest Flow Updating algorithm, abstracting message exchange. . . . 75
3 Flow Updating algorithm, considering concurrent executions at distinct nodes, and abstracting message exchanges. 96
4 Flow Updating with Preferential Grouping algorithm. 103
5 Functions used to decide the leader and compute the reduction potential, relying on the expected variance reduction by including all neighbors. . 106
6 Functions used to decide the leader and compute the reduction potential, relying on the leader average expectation and variance reduction including all neighbors. 106
7 Algorithm to estimate CDF with Flow Updating (FUCDF). 113
8 Auxiliary functions used in the FUCDF algorithm. 114

List of Tables

2.1 Taxonomy of aggregation functions 14

3.1 Taxonomy from a communication perspective. 19
3.2 Taxonomy from a computation perspective. 38

Part I

Exordium

The Purpose of this Work

Chapter 1

Introduction

In the current times, globalization is one of the main keywords that characterizes our world, from economics to scientific innovation, being attached to a new way of thinking and seeing things, and revealing itself as a fundamental instrument for the success of single individuals and collectivities (enterprises, institutions, etc.). Even in the area of computer science and engineering, *global computing* has assumed itself as a topic of great importance in diverse application fields, from entertainment (e.g., internet social networks) to business and R&D (e.g., cloud computing services). In a sense, global computing needs allied with mobile and ubiquitous computing, promote an increased use of large scale distributed systems, which are essential to provide the required support to these computational paradigms.

Considering the need to operate in a distributed fashion, where each peer can have the same opportunity to participate in the system, it becomes difficult to determine any global property, since there is no central element with a global view of the system. Moreover, the knowledge of some global property (e.g., network size; total storage capacity; average load; or majorities) is often required to direct the execution of distributed applications. In such circumstances, data aggregation assumes a core role in the design of distributed systems. In other particular scenarios, like Wireless Sensor Networks (WSN), data collection is often only practicable if aggregation is performed, due to energy constraints found on such environments.

Robbert Van Renesse defined aggregation as "the ability to summarize information", stating that "it is the basis for scalability for many, if not all, large networking services" [124]. In a nutshell, data aggregation is considered a subset of information fusion, aiming at reducing (summarizing) the handled data volume [107]. More pre-

cisely, in this work, we refer to data aggregation as the distributed computation of aggregation functions such as COUNT, AVERAGE or SUM. A more rigorous definition is established in Chapter 2.

Although apparently simple, aggregation has reveled itself as a hard and interesting problem, especially when seeking solutions in distributed settings, where no single element holds a global view of the whole system. In the last decade, several approaches have been proposed to address this problem, revealing different characteristics in terms of accuracy, time and communication tradeoffs. In particular, most of the existing techniques lack fault tolerance, which is often a major concern in distributed and reliable systems. This research is intended to close the existing gap, focusing on the construction of robust aggregation algorithms. The result was the development of a new approach, named *Flow Updating*, with distinctive properties: accurate, fault-tolerant, and able to operate on dynamic settings.

1.1 Motivation

This chapter describe the main motivations of this work, intending to clarify the reader about the importance of aggregation [124] in the design of distributed systems. To this end, several application examples are succinctly described, testifying the need for aggregation. Finally, a brief argumentation will be given to justify the realization of this work, exposing the lack of robust distributed aggregation algorithms.

Aggregation is an essential building block of modern distributed systems, enabling the decentralized computation of meaningful global properties, which can further be used to direct the execution of other distributed applications and algorithms. Aggregation mechanisms also allow the obtention of several networks statics and system information that can be used for administration and monitoring purposes over different decentralized networks, from Peer-to-Peer (P2P) to Wireless Sensor Networks (WSN). For instance, in P2P networks it can be used to obtain useful metrics, such as: the amount of resources available on the network, the average session time of each peer, or the average (maximum or minimum) network load.

In the particular case of WSN, aggregation techniques are essential to monitor and control the covered area, allowing the computation of diverse statistics, such as: the minimum/maximum temperature, the average humidity, measure the concentration of a toxic substance (e.g., carbon monoxide), the noise level, etc. Moreover, due to the

specific constraint found in WSN, data collection is often only practicable if aggregation is performed (especially to optimize and restrain energy consumption).

Often, the data needed by applications correspond to summaries (aggregates) of the collected data, rather than raw data readings. In many applications and algorithms, some system wide metrics are used as input parameters for the execution of basilar operations from which they depend on. An important case is the network size which can be derived from an aggregation algorithm, as many other metrics already referred previously. This metric has revealed itself as fundamental for the correct execution of some algorithms, especially when the actual system size is a priori unknown and can change over time, for example:

- In [96] the estimate of the current network size is used as an essential parameter for the (deterministic or probabilistic) construction of Distributed Hash Tables (DHT) routing topologies, and to adapt the routing structure according to the network dynamism. Other algorithms that construct and maintain a specific routing structure, like DHTs, use the network size estimation to optimize and adjust their performance, namely: Chord [121], Pastry [115], Tapestry [129], Viceroy [93], Symphony [97] and CAN [114].

- Some gossip-based protocols [57; 46] use an estimation of the network size to set the number of targets considered by each node in the information spreading process.

- A probabilistic group communication service for ad-hoc networks based on *random walks* [42; 41], uses the group size estimation (see proposed method in Section 3.1.4) to accelerate self-stabilization. According to the authors, the use of a more accurate estimation of the actual size n, instead of an upper bound N (used to calculate some required parameters), ensures a faster system reaction to network changes (nodes leaving/arriving).

- RaWMS [11], a membership service for wireless ad hoc networks based on random walks, requires the knowledge of the network size to compute the mixing time for a reverse random walk in order to sample peers;

- For the autonomous generation of UIDs (Unique IDentifiers) in mobile environments [70], based on a simple random number generation. The knowledge of

the system size is fundamental to define the minimum length of the generated identifiers, in order to ensure their uniqueness (with high probability).

- The estimation of the network size is required in [2] to set up a probabilistic quorum system in dynamic settings;

Several aggregation algorithms can be found in the actual literature, tailored for different settings, and showing different resulting accuracies, time and communication trade-offs. Some few contributions have compared and evaluated some of the existing algorithms, for instance in [100; 16; 21]. Existing aggregation approaches have proven to be hard to simultaneously obtain accurate results (almost exact), efficiently handle the system dynamism and tolerate faults. In fact, most of the existing techniques do not address fault tolerance, only few have recently exhibited some practical concerns about aggregation robustness, like in [128] that supports discontinuous failures of adjacent nodes within a sort time period. To the best of our knowledge, an efficient and accurate aggregation algorithm with effective robustness capabilities was still missing. According to this observation, and without forgetting some important requirements like: accuracy, scalability, response time, communication efficiency, fault tolerance and adaptability; The need to define a better (more robust) solution to the aggregation problem represented the main motivation for the development of this research study.

1.2 Contributions

The main contribution of this research stands in the development of a novel and robust aggregation approach, tolerant to message loss and that can operate in dynamic environments, supporting churn (i.e., nodes leaving/arriving) and changes of the initial input values. The designed aggregation technique, *Flow Updating*, possesses important attributes: it is accurate, tolerates faults (message loss and node crashes), and it is able to self-adapt to network and value changes. Due to these unique characteristics which distinguish it from other existing aggregation algorithms, enabling it's practical application in realistic settings, *Flow Updating* constitutes a valuable contribution to the area of distributed computing. Beside this, other contributions can be found in this research work, namely: a survey and taxonomy of the state of the art on distributed data aggregation algorithms; study of the dependability issues of the existing aggregation algorithms; a novel scheme to estimate the statistic distribution of an attribute,

more precisely its Cumulative Distributed Function (CDF), based on *Flow Updating*.

Up to now, the contributions from this research work were materialized in the following publications:

- "Fault-Tolerant Aggregation by Flow Updating", the 9th IFIP International Conference on Distributed Applications and interoperable Systems (DAIS), 2009 [72];
- "Dependability in Aggregation by Averaging", Simpósio de Informática (INForum), 2009 [71];
- "Fault-Tolerant Aggregation for Dynamic Networks", 29th IEEE Symposium on Reliable Distributed Systems (SRDS), 2010 [74];
- "Estimativa Contínua e Tolerante a Faltas de Funções Distribuição Cumulativa em Redes de Larga Escala", Simpósio de Informática (INForum), 2011 [17];

Moreover, the designed technique also contributed to the development of other ideas, and provided a relevant collaboration in other papers:

- "Extrema Propagation: Fast Distributed Estimation of Sums and Network Sizes", IEEE Transactions on Parallel and Distributed Systems (In Preprint) [9];
- "Fault-Tolerant Aggregation: Flow Update Meets Mass Distribution", 15th International Conference On Principles Of Distributed Systems – OPODIS 2011 (Accepted) [3];

Nonetheless, more publications are excepted to come from this work. For instance, a few more are already under submission (waiting for review):

- "A Survey on Distributed Data Aggregation Algorithms" (submitted to ACM Computing Surveys);
- "*Flow Updating*: Fault-Tolerant Aggregation for Dynamic Networks" (submitted to IEEE Transactions on Parallel and Distributed Systems);

1.3 Organization

This section describes the organization of the contents of this research work. The concept of aggregation function is defined in Chapter 2, clarifying the reader about the

properties of the function that are intended to be computed in a distributed way. The previous distributed aggregation algorithms are surveyed and categorized in Chapter 3, and their dependability issues are revealed in Chapter 4. The main contribution of this work, a novel approach designated *Flow Updating*, is detailed in Chapter 5, providing a correctness analysis and some additional practical considerations to allow its application on real settings. Additionally, a novel approach based on Flow Updating is introduced in Chapter 6, allowing the estimation of the Cumulative Distribution Function (CDF) of global attributes. *Flow Updating* is empirical evaluated in Chapter 7, describing the simulation settings and discussing the results obtained on several scenarios. Finally, Chapter 8 provides some concluding remark, summarizing the main results and contributions, and pointing out some future research directions.

Chapter 2

Problem Definition

This chapter defines the concept of aggregation and the problem addressed by this research work. Additionally, the main properties of an aggregation function are defined, and a taxonomy of the most common aggregations functions is provided.

In a nutshell, aggregation can be simply defined as "the ability to summarize information", quoting Robbert Van Renesse [124]. Data aggregation is considered a subset of information fusion, aiming at reducing (summarize) the handled data volume [107]. Here, we provide a more precise definition, and consider that the process consists in the computation of an *aggregation function* defined by:

Definition 2.1 (Aggregation Function)**.** *An aggregation function f takes a multiset of elements from a domain I and produces an output of a domain O.*

$$f : \mathbb{N}^I \to O$$

The input being a multiset emphasizes that: first, the order in which the elements are aggregated is irrelevant; second, a given value may occur several times. Frequently, for common aggregation functions such as *min*, *max*, and *sum*, both I and O are of the same domain. For others, such as *count* (which gives the cardinality of the multiset), the result is a nonnegative integer ($\mathbb{N}$), regardless of the input domain.

An aggregation function aims to summarize information. Therefore, the result of an aggregation (in the output domain O) typically takes much less space than the multiset to be aggregated (less than the elements from $\mathbb{N}^I$). We will leave unspecified what is acceptable for some function to be considered as summarizing information, and therefore an aggregation function. Nonetheless, it can be said that the output domain O

is not usually a multiset (in general, $O \neq \mathbb{N}^I$), and that the identity function is clearly not an aggregation function as it definitely does not summarize information.

2.1 Properties of Aggregation Functions

When trying to design a distributed algorithm to compute an aggregation function (e.g., COUNT, SUM, AVERAGE, MIN, MAX, or RANGE), it is easy to observe than some functions might be more difficult to compute than others, raising additional concerns in order to obtain the correct result. For instance, the MIN and MAX can be trivially computed in a distributed way, simply applying the MIN/MAX to any received inputs at any (intermediary) node, and further propagating the result, independently from the order of inputs and repetition. On the contrary, to compute the COUNT or the SUM one must be careful not to account the same value twice (at intermediary nodes). Also, in the case of RANGE one must wait for the determination of the MIN and the MAX before calculating the result (difference between them).

Analyzing more carefully this examples, one can suspect the existence of some relation between the difficulty by distributedly computing an aggregation function, and its characteristics in terms of decomposability and duplicate sensitivity. For this reason, and in order to refine the concept of aggregation function, some properties are formally defined.

2.1.1 Decomposability

For some aggregation functions, we may need to perform a single computation involving all elements in the multiset to calculate the result. However, for many cases, one needs to avoid such centralized computation. In order to perform distributed *in-network* aggregation[1], it is relevant whether or not the aggregation function can be *decomposed* into several computations, involving sub-multisets of the multiset to be aggregated. Therefore, for distributed aggregation it is useful to define the notion of *decomposable aggregation function*, and a subset of it which we call *self-decomposable aggregation functions*.

[1] A definition of in-network aggregation is proposed in [51]. Here, we consider the term *in-network* in a more general way, referring solely to the need to process information at intermediate nodes in a common network.

Definition 2.2 (Self-decomposable Aggregation Function)**.** *An aggregation function $f : \mathbb{N}^I \to O$ is said to be self-decomposable, if for some (merge) operator $\diamond$ and all non-empty multisets X and Y:*

$$f(X \uplus Y) = f(X) \diamond f(Y),$$

where $\uplus$ denotes the standard multiset sum (see, e.g.,[122]).

According to the above definition, the operator $\diamond$ is commutative and associative, given that the aggregation result is the same for all possible partitions of a multiset into sub-multisets. This can be very handy in terms of distributed computation, allowing self-decomposable aggregation functions to be easily separated into several aggregation processes, as illustrated by Figure 2.1. As example, many traditional functions such as MIN, MAX, SUM and COUNT are self-decomposable:

$$\begin{aligned} \text{sum}\,(\{x\}) &= x, \\ \text{sum}(X \uplus Y) &= \text{sum}(X) + \text{sum}(Y). \end{aligned}$$

$$\begin{aligned} \text{count}(\{x\}) &= 1, \\ \text{count}(X \uplus Y) &= \text{count}(X) + \text{count}(Y). \end{aligned}$$

$$\begin{aligned} \min(\{x\}) &= x, \\ \min(X \uplus Y) &= \min(X) \sqcap \min(Y). \end{aligned}$$

where $\sqcap$ is the *meet* operator which for natural numbers coincides with the minimum (i.e., $x \sqcap y = min(\{x, y\})$) [32].

Definition 2.3 (Decomposable Aggregation Function)**.** *An aggregation function $f : \mathbb{N}^I \to O$ is said to be decomposable, if for some function g and a self-decomposable aggregation function h:*

$$f = g \circ h$$

i.e, for any non-empty multiset X, it can be expressed as:

$$f(X) = g(h(X))$$

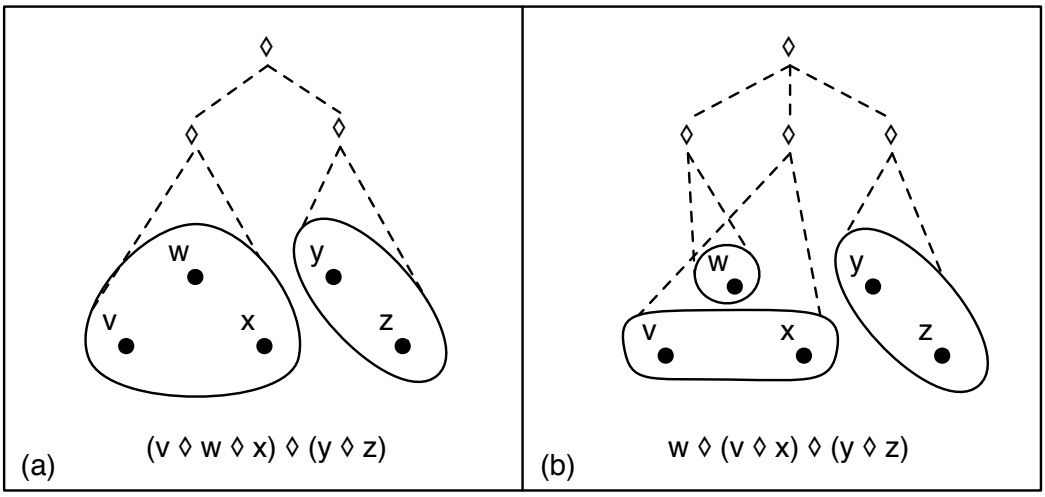


Figure 2.1: Examples of computation of a self-decomposable aggregation function (associative and commutative).

In other words, this means that a decomposable aggregation function f results from the composition of a function g with a self-decomposable aggregation function h. From this definition, self-decomposable aggregation functions are a subset of decomposable functions, where $g = \text{id}$, the identity function. This means, that while for self-decomposable functions the *intermediate* results (e.g., for in-network aggregation) are computed in the output domain O, for a general decomposable function, we may need a different auxiliary domain to hold the intermediate results.

A classic example of a decomposable (but not self-decomposable) aggregation function is AVERAGE, which can be expressed in the following way:

$$\begin{aligned} \text{average}(X) &= g(h(X)), \qquad \text{with} \\ h(\{x\}) &= (x, 1) \\ h(X \uplus Y) &= h(X) + h(Y), \\ g((s, c)) &= s/c, \end{aligned}$$

where h is a self-decomposable aggregation function that outputs values of an auxiliary domain (pairs of values) and $+$ is the standard pointwise sum of pairs (i.e., $(x_1, y_1) + (x_2, y_2) = (x_1 + x_2, y_1 + y_2)$). Another example is the RANGE function in statistics, which gives the difference between the maximum and the minimum value:

$$\begin{aligned}
\text{range}(X) &= g(h(X)), \qquad \text{with} \\
h(\{x\}) &= (x, x) \\
h(X \uplus Y) &= h(X) \diamond h(Y), \\
g((d, u)) &= u - d, \qquad \text{given that} \\
(x_1, x_2) \diamond (y_1, y_2) &= (\min(\{x_1, y_1\}), \max(\{x_2, y_2\})).
\end{aligned}$$

2.1.2 Duplicate sensitiveness and idempotence

Depending on the aggregation function, it may be relevant whether a given value occurs several times in the multiset. For some aggregation functions, such as MIN and MAX, the presence of duplicate values in the multiset does not influence the result, which only depends on its *support set* (i.e., the set obtained by removing all duplicates from the original multiset). For example:

$$\min(\{1, 3, 1, 2, 4, 5, 4, 5\}) = \min(\{1, 3, 2, 4, 5\}) = 1$$

For others aggregation functions, like SUM and COUNT, the number of times each element occurs (i.e., *multiplicity*) is relevant. For example:

$$8 = \text{count}(\{1, 3, 1, 2, 4, 5, 4, 5\}) \neq \text{count}(\{1, 3, 2, 4, 5\}) = 5$$

Therefore, duplicate sensitiveness is important and must be taken into account for distributed aggregation, in order to compute the correct result.

Definition 2.4 (Duplicate Insensitive Aggregation Function)**.** *An aggregation function $f : \mathbb{N}^I \to O$ is said to be duplicate insensitive, if for any non empty multiset M and its corresponding support set S:*

$$F(M) = f(S).$$

Moreover, some duplicate insensitive functions (like MIN and MAX) can be implemented using an idempotent binary operator, that can be successively applied (by intermediate nodes) on the elements of the multiset (any number of times without affecting the result). This helps in obtaining fault tolerance and decentralized processing,

	Decomposable		Non-decomposable
	Self-decomposable		
Duplicate insensitive	MIN, MAX	RANGE	DISTINCT COUNT
Duplicate sensitive	SUM, COUNT	AVERAGE	MEDIAN, MODE, CDF

Table 2.1: Taxonomy of aggregation functions

allowing retransmissions or sending values across multiple paths. Unfortunately, the distributed application of an idempotent operator is not always possible, even for some duplicate insensitive aggregation functions, such as DISTINCT COUNT (i.e., cardinality of the support set). In fact, the application of an idempotent operator in a distributed way to compute an aggregation function is only possible, if the function is duplicate insensitive and self-decomposable.

2.2 Taxonomy of common aggregation functions

Building on the concepts of decomposability and duplicate sensitiveness, we can obtain a taxonomy of aggregation functions (Table 2.1). This table helps to clarify how suited an aggregation function is to a distributed computation. Non-decomposable aggregation functions are harder to compute than decomposable ones, being vulgarly labeled in the literature as “complex”. Commonly, duplicate insensitive aggregation functions are easier to compute than duplicate sensitive. As we will see in the next chapter, one way to obtain fault-tolerance (at the cost of some accuracy) is to use duplicate insensitive approaches to estimate some aggregation function (e.g., *sketches*), replacing the use of non-idempotent operations (like sum) by idempotent ones (like max).

Part II

State of the Art

Chapter 3

Related Work

The body of knowledge on data aggregation in composed by a considerable amount of distinct approaches, that allow the distributed computation of aggregation functions. This chapter surveys a wide range of existing algorithms, identifying and classifying the most important techniques, and discussing their advantages/disadvantage in terms of communication and computational complexity.

Some surveys about aggregation can already found in the literature, but they focus specifically on techniques for WSN [51; 113; 117; 6; 107]. Namely, several in-network aggregation techniques for WSN are depicted in [51], typically operating at the network level and needing to deal with the resource constraints of sensor nodes (limited computational power, storage and energy resources). A review more focused on energy efficiency is presented in [113], and on security in [117; 6]. Finally, a state-of-the-art on information fusion techniques for WSN is available in [107], covering a broad view of the sensor fusion process, considering data aggregation as a subset of information fusion. Here, data aggregation algorithms are addressed at a higher abstraction level, providing a comprehensive and more generic view of the problem independently from the type of network used.

The most relevant distributed data aggregation algorithms are succinctly detailed in this section, and simple taxonomies are proposed, classifying them according to two main perspectives: communication and computation (see Table 3.1 and 3.2). The first viewpoint refers to the routing protocols and intrinsic network topologies used to support the existing aggregation techniques. The second perspective points out the aggregation functions computed by the algorithms and the main principles from which they are based on. Other perspectives (e.g., algorithm requirements, or covered types

of aggregation functions) could have been considered, since the mapping between the algorithms attributes is multidimensional. However, we believe that the two chosen perspectives will provide a clear presentation and understanding of the current state of the art.

3.1 Communication

Three major classes of aggregation algorithms are identified from the communication perspective, according to the characteristics of their communication pattern (routing protocol) and network topology: *structured* (usually, hierarchy-based), *unstructured* (usually, gossip-based), and *hybrid* (mixing the previous categories).

The *structured* communication class refers to aggregation algorithms that are dependent on a specific network topology and routing scheme to operate correctly. If the required routing topology is not available, then an additional preprocessing phase is needed in order to create it, before starting the execution of the algorithm. This dependency limits the use of these techniques in dynamic environments. For instance, in mobile networks these algorithms need to be able to continuously adapt their routing structure to follow network changes. Typically, algorithms are directly affected by problems from the used routing structure. For example, in tree-based communication structures a single point of failure (node/link) can compromise the delivery of data from all its subtrees, and consequently impair the applications supported by that structure. In practice, hierarchical communication structures (e.g., tree routing topology) are the most often used to perform data aggregation, especially in WSN. Alternative routing topologies have also been considered, like the ring topology, although very few approaches rely on it.

The *unstructured* communication category covers aggregation algorithms that can operate independently from the network organization and structure, without establishing any predefined topology. In terms of communication, this kind of algorithms is essentially characterized by the used communication pattern: *flooding/broadcast*, *random walk* and *gossip*. The *flooding/broadcast* communication patterns is associated to the dissemination of data from one node to all the network or group of nodes – "one to all". A *random walk* consists in sequential message transmissions, from one node to another – "one to one". The *gossip* communication pattern refers to a well known communication protocol, based on the spreading of a rumor [111] (or an epidemic dis-

	Routing	Algorithms
Structured	*Hierarchy (tree, cluster, multipath)*	TAG [92], DAG [106], I-LEAG [16], Sketches [29], RIA-LC/DC [49][50], Tributary-Delta [95], Q-Digest[120]
	Ring	(Horowitz and Malkhi, 2003) [65]
Unstructured	*Flooding/Broadcast*	Randomized Reports [13]
	Random walk	Random Tour [98], Sample & Collide [56; 98], Capture-Recapture [94]
	Gossip	Push-Sum Protocol [80], Push-Pull Gossiping [67; 103; 68], DRG [21], Extrema Propagation [8] Equi-Depth[61], Adam2 [116] Hop-Sampling [83; 84], Interval Density [83; 84]
Hybrid	*Hierarchy + Gossip*	(Chitnis et al., 2008) [23]

Table 3.1: Taxonomy from a communication perspective.

ease), in which messages are sent successively from one node to a selected number of peers – "one to many". In the recent years, several aggregation algorithms based on gossip communication have been proposed, in an attempt to take advantages of its simplicity, scalability and robustness.

The *hybrid* class groups algorithms that mix the use of different routing strategies from the previous categories, with the objective to combine their virtues and reduce their weakness, in order to obtain an improved aggregation approach.

The different communication structures and patterns are summarized in the taxonomy proposed by Table 3.1, associating each routing class to some pertinent algorithms. More details about the identified classes are given hereafter.

3.1.1 Hierarchy-based

Traditionally, existing aggregation algorithms operate on a hierarchy-based communication scheme. Hierarchy-based approaches are often used to perform data aggregations, especially in WSN. This routing strategy consists on the definition of a hierarchical communication structure (e.g., spanning tree), rooted at a single node, commonly designated as *sink*. In general, in a hierarchy-based approach the data is simply disseminated from level to level, up the hierarchy, in response to a query request made by the sink, which computes the final result. Besides the sink, other special nodes can be defined to compute intermediate aggregates, working as aggregation points that forward their results to upper level nodes until the sink is reached. Aggregation algorithms based on hierarchic communication usually work in two phases, involving the participation of all nodes in each one: *request* phase and *response* phase. The *request* phase corresponds to the spreading of an aggregation request throughout all the network. Several considerations must be taken into account before starting this phase, depending on which node wants to performs the request and on the existing routing topology. For instance: if the routing structure has not been established yet, it must be created and ought to be rooted at the requesting node; if the required topology is already established, first the node must forward its request to the root, in order to be spread (from the sink) across all the network. During the *response* phase, all the nodes answer the aggregation query by sending the requested data toward the sink. In this phase, nodes can be asked to simply forward the received data or to compute partial intermediate aggregates to be sent.

The aggregation structure of hierarchy-based approaches provides a simple strategy, that enables the exact computation of most aggregation functions (without failures), in an efficient manner in terms of energy consumption. However, in adverse environments this type of approach exhibits some fragility in terms of robustness, since a single point of failure can jeopardize the obtained result (losing the subtree data). Furthermore, to correctly operate in dynamic environments, where the network continuously changes (nodes joining/leaving), extra resources are required to maintain an updated routing structure. Next, some important approaches from this class are described.

TAG The Tiny AGgregation service for ad-hoc sensor networks described by Madden et al. [92] represents a classical tree-based in-network aggregation approach. As

referred by the authors, in a sense TAG is agnostic to the implementation of the tree-based routing protocol, as far as it satisfies two important requirements. First, it must be able to deliver query requests to all the network nodes. Second, it must provide at least one route from every node (that participates in the aggregation process) to the sink, guaranteeing that no duplicates are generated (at most one copy of every message must arrive). This algorithm requires the previous creation of a tree-based routing topology, and also the continuous maintenance of such routing structure in order to operate over mobile networks.

TAG supplies an aggregation service inspired in the selection and aggregation features of database query languages, providing a declarative SQL-like (Structured Query Language) query language to the users. This algorithm offers grouping capabilities and implements basic database aggregation functions, among others, such as: COUNT, MAXIMUM, MINIMUM, SUM and AVERAGE. The aggregation process consists of two phases: a *distribution phase* (in which, the aggregation query is propagated along the tree routing topology, from the root to the leaves) and a *collection phase* (where the values are aggregated from the children to the parents, until the root is reached). The obtention of the aggregation result at the root incurs a minimum time overhead that is proportional to the tree depth. This waiting time is needed to ensure the conclusion of the two execution phases and the participation of all nodes in the aggregation process.

A *pipelined aggregate* technique (detailed in [91]) has been proposed to minimize the effect of the waiting time overhead. According to this technique, smaller time intervals (relatively to the overall needed time) are used to repetitively produce periodic (partial) aggregation results. In each time interval, all nodes that have received the aggregation request will transmit a partial result, which is calculated from the application of the aggregation function to their local reading and to the results received from their children in the previous interval. Along time, after each successive time interval, the aggregated value will result from the participation of a growing number of nodes, increasing the reliability and accuracy of the result, becoming close to the correct value at each step. The correct aggregation result should be reached after a minimum number of iterations (in an ideal fail-safe environment).

Following the authors concerns about power consumption, additional optimization techniques were proposed to the TAG basic approach, in order to reduce the number of messages sent, taking advantages of the shared communication medium in wireless networks (which enables message snooping and broadcast) and giving decision power

to nodes. They proposed a technique called *hypothesis testing*, where each node can decide to transmit the value resulting from its subtree, only if it will contribute to the final result.

DAG An aggregation scheme for WSN based on the creation of a DAG (Directed Acyclic Graph) is proposed in [106]. The objective was to reduce the effect of message loss of common tree-based approaches by allowing nodes to possess alternative parents. The DAG is created by setting multiple parents (within radio range) to each node, as its next hop toward the sink. In more detail, request messages are extended with a list of parent nodes (IDs), enabling children to learn the parent's parent (grandparents) which are two hops away. In order to avoid duplicated aggregates, only a parent is chosen to aggregate intermediate values, preferably a common parent of its parents. The most common parent's parent between the list received from parents is chosen as the destination aggregator, otherwise one of the parents is chosen (e.g., when a node has only one parent node). Response messages are handled according to specific rules to avoid duplicate processing: they can be aggregated, forwarded or discarded. Messages are aggregated if the receiving node corresponds to the destination, forwarded if the destination is a node's parent, and discarded otherwise (destination is not the node or one of its parents). Note that, although the same message can be duplicated and multiple "copies" can reach the same node (a grandparent), they will have the same destination node and only one of them (from the same source) will be considered for aggregation (after receiving all messages from children/grandchildren).

This method takes advantage of the path redundancy introduced by the use of multiple parents to improve the robustness of the aggregation scheme (tolerance to message loss), when compared to traditional tree-based techniques. Though a better accuracy can be achieved, it comes at the cost of an higher energy consumption, as more messages with an increased size are transmitted. Note that this approach does not fully overcome the message loss problem of tree routing topologies, as some nodes may have a single parent, being dependent from the quality of the created DAG.

Sketches An alternative multi-path based approach is proposed in [29] to perform in-network aggregation for sensor databases, using small sketches. The defined scheme is able to deal with duplicated data upon multi-path routing and compute duplicate-sensitive aggregates, like COUNT, SUM and AVERAGE. This algorithm is based on the probabilistic counting sketches technique introduced by Flajolet and Martin [52] (FM),

used to estimate the number of distinct elements in a data collection. A generalization of this technique is proposed to be applied to duplicate-sensitive aggregation functions (non-idempotent), namely the SUM. The authors consider the use of multi-path routing to support communication failures (links and nodes), providing several possible paths to reach a destination. Like common tree-based approaches, the algorithm consists of two phases: first, the sink propagates the aggregation request across the whole network; second, the local values are collected and aggregated along a multi-path structure from the children to the root. In this particular case, during the request propagation phase, all nodes compute their distance (level) to the root and store the level of their neighbors, establishing a hierarchical multi-path routing topology (similar to the creation of multiple routing trees). In the second phase, partial aggregates are computed across the routing structure, using the adapted counting sketch scheme, and sent to the upper levels in successive rounds. Each round corresponds to a hierarchy level, in which the received sketches from children nodes are combined with the local one, until the sink is reached. In the last round, the sink merges the sketches of its neighbors and produces the final result, applying an estimation function over the sketch. Notice that the use of an auxiliary structure to summarize all data values (FM sketches), and correspondent estimator, will introduce an approximation error that will be reflected in the final result. However, according to this aggregation scheme, it is expected that data losses (mitigated with the introduction of multiple alternative paths) will have an higher impact in the result accuracy than the approximation error introduced by the use of sketches (to handle duplicates).

I-LEAG This cluster-based aggregation approach, designated I-LEAG [16] (Instance-Local Efficient Aggregation on Graphs), requires the pre-construction of a different routing structure – *Local Partition Hierarchy*, which can be viewed as a logical tree of local routing partitions. The routing structure is composed by a hierarchy of clusters (partitions), with upper level clusters comprising lower level ones. A single pivot is assigned to each cluster, and the root of the tree corresponds to the pivot of the highest level cluster (that includes all the network graph). This algorithm emphasizes local computation to perform aggregation, being executed along several sequential phases. Each phase, correspond to a level of the hierarchy, in which the algorithm is executed in parallel by all clusters of the corresponding level (from lower levels to upper levels). Basically, the algorithm proceeds as follow: each cluster checks for local conflicts

(different aggregation outputs between neighbors); detected conflicts are reported to pivots, which compute the new aggregated value and multicast the result to the cluster; additionally, every node forwards the received result to all neighbors that do not belong to the cluster; received values are used to update the local aggregation value (if received from a node in the current cluster) or to update neighbor aggregation output (if received from a neighbor of the upper level cluster), enabling the local detection of further conflicts. Conflicts are only detected between neighbors that belong to a different clusters in the previous phase, with different aggregation outputs from those clusters. A timer is needed to ensure that all messages sent during some phase reach their destination by the end of the same phase. Further, two extension of the algorithm were proposed to continuously compute aggregates over a fixed network where node inputs may change along time: MultI-LEAG and DynI-LEAG [15]. MultI-LEAG mainly corresponds to consecutive executions of I-LEAG, improved to avoid sending messages when no input changes are detected. Inputs are sampled at regular time intervals and the result of the current sampling interval is produced before the next one starts. DynI-LEAG concurrently execute several instances of MultI-LEAG, pipelining its phases (ensuring that every partition level only executes a single MultI-LEAG phase at a time), and more frequently sampling inputs to faster track changes but at the cost of a higher message complexity. Despite the authors effort to efficiently perform aggregation, these algorithms are restricted to static networks (with fixed size), without considering the occurrence of faults.

Tributary-Delta This approach mixes the use of tree and multi-path routing schemes to perform data aggregation, combining the advantages of both to provide a better accuracy in the presence of communication failures [95]. Two different routing regions are defined: *tributary* (tree routing, in analogy to the shape formed by rivers flowing into a main stem) and *delta* (multi-path routing, in analogy to the landform of a river flowing into the sea). The idea is to use tributaries in regions with low message loss rates to take advantage of the energy-efficiency and accuracy of a traditional tree-based aggregation scheme, and use deltas in zones where message losses have a higher rate and impact (e.g., close to the sink where messages carry values corresponding to several node readings) to benefit from the multi-path redundancy of sketch based schemes. Two adaptation strategies (TD-Coarse and TD) are proposed to shrink or expand the delta region, according to the network conditions and a minimum percentage of con-

tributing nodes predefined by the user. The prior knowledge of the network size is required, and the number of contributing nodes needs to be counted (or count the non contributing nodes in a tributary subtree), in order to estimate the current participation percentage. Conversion functions are also required to convert partial results from the tributary (tree-based aggregation) into valid inputs to be used in the delta region (by the multi-path algorithm). Experimental results applying TAG [92] in tributaries and Synopses Diffusion [108] (see Section 3.2.3) in deltas, showed that this hybrid approach performs better when compared to both aggregation algorithms used separately.

Other approaches Several other hierarchy-based aggregation approaches can be found in the literature, most of them differing somehow on the supporting routing structure, or on the way it is built. Beside alternative variations of the hierarchic routing topology, some optimization techniques to the aggregation process can also be found, especially to reduce the energy-consumption in WSN.

In [86] an aggregation scheme over DHTs (Distributed Hash Tables) is proposed. This approach is characterized by its tree construction protocol, that use a *parental function* to map a unique parent to each node, building an aggregation tree in a bottom-up fashion (unlike traditional approaches). The authors consider the coexistence of multiple trees to increase the robustness of the algorithm against faults, as well as the continuous execution of a tree maintenance protocol to handle the dynamic arrival and departure of nodes. Two operation modes are proposed to perform data aggregation (and data broadcast): *default* and *on-demand*. In the *default* mode, the algorithm is executed in background, taking advantage of messages exchanged by the tree maintenance protocol (appending some additional information to these messages). The *on-demand* mode corresponds to the traditional aggregation scheme found on tree-based algorithms.

Zhao et al. [130] proposed an approach to continuously compute aggregates in WSN, for monitoring purposes. They assume that the network continuously computes several aggregates, from which at least one corresponds to the minimum/maximum – computed using a simple diffusion scheme. A tree is implicitly constructed during the diffusion process (node with the min/max value is set as the root of the created tree) and is used for the computation of other aggregates (e.g., average and count). In practice, two different schemes are used: a *digest diffusion* algorithm to compute idempotent aggregates which is used to construct an aggregation tree, and a *tree di-*

gest scheme similar to common hierarchy-based approaches that operates over the tree routing structure created by the previous technique.

Alternative hierarchic routing structures are found in the literature to support aggregation, namely: a BFS (Breadth First Search) tree is used in the GAP (Generic Aggregation Protocol) [30] protocol to continuously compute aggregates for network management purposes; the creation of a GIST (Group-Independent Spanning Tree) based on the geographic distribution of sensors is described in [75], taking into consideration the variation of the group of sensors that may answer an aggregation query. A previous group-aware optimization technique has been proposed: GaNC (Group-Aware Network Configuration) [119]. GaNC influences the routing tree construction by enabling nodes to preferably set parents from the same group (analyzing the GROUP BY clause of the received aggregation queries) and according to a maximum communication range, in order to decrease message size and consequently reduce energy consumption. Some algorithms [101; 87; 126] based on swarm intelligence techniques, more precisely ant colony optimization, can also be found in the literature to construct optimal aggregation trees, once more to improve the energy efficiency of WSN. Ant colony optimization algorithms are inspired in the foraging behavior of ants, leaving pheromone trails that enable others to find the shortest path to food. In this kind of approach, the aggregation structure is iteratively constructed by artificial ant agents, consisting in the paths (from different sources to the sink) with the higher pheromone values, and where nodes that belong to more than one path act as aggregation points.

Some studies [89; 62] have shown that deciding which node should act as a data aggregator or forwarder has an important impact on the energy-consumption and lifetime of WSN. A routing algorithm, designated AFST (Adaptive Fusion Steiner Tree), that adaptively decides which nodes should fuse (aggregate) data or simply forward it is described in [89]. AFST evaluates the cost of data fusion and transmission, during the construction of the routing structure in order to minimize energy consumption of data gathering. A further extension to this scheme was proposed to handle node arrival/departure, Online AFST [88], with the objective of minimizing the cost and impact of dynamism in the routing structure. In LEACH (Low-Energy Adaptive Clustering Hierarchy) [62; 63], a cluster-based routing protocol for data gathering in WSN, the random rotation of cluster-heads along time is proposed in order to distribute the energy consumption burden of collecting and fusing (compressing) cluster's data.

Filtering strategies can also be applied to reduce energy consumption in hierarchy-

based aggregation approach. For instance, A-GAP [112] is an extension of GAP (previously referred) which uses filters to provide a controllable accuracy of the protocol. Local filters are added at each node in order to control whether or not an update is sent. Updates are discarded according to a predefined accuracy objective, resulting in a reduction in terms of communication overhead (number of messages). Filters can dynamically adjust along the execution of the protocol, allowing the control of the trade-off between accuracy and overhead. Another similar approach to reduce message transmissions according to a tolerated error value is proposed in [33], adaptively adjusting filters according to a Potential Gains Adjustment (PGA) strategy. A framework called TiNA (Temporal coherency-aware in-Network Aggregation) that filters reported sensor readings according to their temporal coherency was proposed in [119]. This framework operates on the top of existing hierarchic-based aggregation schemes like TAG. In particular, TiNA defines an additional TOLERANCE clause to allow users to specify the desired temporal coherency tolerance of each aggregation query, and filter the reported sensor data (i.e., readings within the range of the specified value are suppressed).

3.1.2 Ring

Very few aggregation approaches are supported by a ring communication structure. This particular type of routing topology is typically surpassed by hierarchic ones, which are used instead. For instance, the effect of failures in a ring can be worst than on hierarchic topologies, as a single point of failure can break the all communication chain. Furthermore, the time complexity of rings to propagate data across all the network is typically higher, providing a slower data dissemination. However, this kind of topology can be explored in alternative ways, that can in some sense circumvent the aforementioned limitations.

It is worth referring to an alternative approach described by Horowitz and Malkhi [65], based on the creation of a virtual ring to obtain an estimation of the network size (COUNT) at each node. This technique relies solely on the departure and arrival of nodes to estimate the network size, without requiring any additional communication. Each node of the network holds a single successor link, forming a virtual ring. It is assumed that each node possess an accurate estimator. Upon the arrival of a new node, a random successor among the existing nodes, named *contact point*, is assigned to it. During the joining process, the new node gets the *contact point* estimator and incre-

ments it (by one). At the end of the joining process, the two nodes (joining node and *contact point*) will yield the new count estimate. Upon the detection of a departure, the inverse process is executed. This method provides a disperse estimative over the whole network, with an excepted accuracy that ranges from $n/2$ to n^2, where n represent the correct network size [1]. Despite the achieved low accuracy and considerable result dispersion, this algorithm has a substantially low communication cost (i.e., communicates only upon arrival/departure, without any further information dissemination; each joining node communicates only with two nodes).

3.1.3 Flooding/Broadcast

Flooding/Broadcast based approaches promote the participation of all network nodes in the data aggregation process. The information is propagated from a single node (usually a special one) to the whole network, sending messages to all neighbors – "one to all". This communication pattern normally induces a high network load, during the aggregation process, implying in some cases a certain degree of centralization of data exchanges. Tree-based approaches are a traditional example of use of this communication pattern, but in this case supported by a hierarchic routing topology. Additional examples which are not sustained by any specific structured routing topology are described below.

Randomized Reports A naive algorithm to perform aggregation, will consist of broadcasting a request to the whole network (independently from the existing routing topology), collect the value at all nodes and compute the result at the starting node. This will likely lead to network congestion and an expected overload of the source node, due to *feedback implosion*. However, a predefined response probability could be used to mitigate this drawback, such that network nodes will only decide to respond according to the defined probability. Such *probabilistic polling* method was proposed in [13] to estimate the network size. The source node broadcast a query request with a sampling probability p, that will be used by all remaining nodes to decide whether to reply or not. All the received responses will be counted by the querying node (during a predefined time interval), knowing that it will receive a total number of replies r according to the given probability. At the end, the network size $\widehat{n}$ can be estimated at the source by $\widehat{n} = r/p$.

[1] Along this work, we will always denote n as the network size, unless explicitly indicated otherwise.

Other Approaches A similar approach based on the same principle (sampling probability) is proposed in [76], to approximate the size of a single-hop radio network, considering the occurrence of collisions (i.e., a transmission succeeds if exactly a single node chooses to send a message). The algorithm proceeds along several phases, counting the number of successful transmissions until it approximates an expected value (based on a probabilistic observation), in order to estimate the network size. In each consecutive phase, different values are set at each node for the probability to decide to send a message (decrementing), and the number of performed trials (incrementing). Further improvements to this approach have been proposed in [78], aiming at making it immune against adversary attacks.

3.1.4 Random Walk

Random walk based approaches are usually associated to a data sampling process to further estimate an aggregation value, involving only a partial amount of network nodes. Basically, this communication process consists on the random circulation of a token. A message is sequentially sent from one node to another randomly selected neighbor – "one to one", until a predefined stopping criteria is met (e.g., maximum number of hops, reach a selected node or return to the initial one). Usually, a small amount of messages are exchanged in this kind of approach, since only a portion of the network is involved in the aggregation process. Due to the partial participation of the network, algorithms using this communication pattern normally rely on probabilistic methods to produce an approximation of the computed aggregation function. Probabilistic methods provide estimations of the result with a known bounded error. If the execution conditions and the considered parameters of the algorithm are maintained, the estimation error is expected to be maintained (with constant bounds) along time. This kind of aggregation algorithms will not converge to the correct aggregation value, and the result will always contain an estimation error.

Random Tour The random tour approach [98] is based on the execution of a random walk to estimate a sum of functions of the network nodes, $\Phi = \sum_{i \in \mathcal{N}} \phi(i)$, for a generic function $\phi(i)$ where i denotes a node and $\mathcal{N}$ the set of nodes (e.g., to estimate the network size, count: $\phi(i) = 1$, for all $i \in \mathcal{N}$). The estimate is computed from the accumulation of local statistics into a initial message, all of which are gathered during a random walk, from the originator node until the message returns to it. The

initiator node i initializes a variable X with the value $\phi(i)/d_i$ (where d_i denotes the degree of node i, i.e., number of adjacent nodes). Upon receive, each node j adds to X by $\phi(j)/d_j$ (i.e., $X \leftarrow X + \phi(j)/d_j$). In each iteration, the message tagged with X is updated and forwarded to a neighbor, chosen uniformly at random, until it returns to the initial node. When the originator receives back the message originally sent, it computes the estimate $\widehat{\Phi}$ (of the sum Φ) by $\widehat{\Phi} = d_i X$.

Other approaches Other approaches based on random walks can be found in the literature, but they are commonly tailored for specific settings and to the computation of specific aggregation functions, like COUNT (to estimate the network or group size).

For instance, to accelerate self-stabilization in a group communication system for ad-hoc networks, a scheme to estimate the group size based on random walks is proposed in [42] (first published in [41]). In this specific case, a mobile agent (called *scouter*) performs a random walk and collects information about alive nodes to further estimate the system size. The agent carries the set of all visited nodes and a counter associated to each one of them. Whenever the agent moves to a node, all the counters are incremented by one except the one of the current node, which is set to zero. Large counter values are associated to nodes that have been less recently visited by the *scouter*, becoming more likely to be suspected of nonexistence. Counters are bounded by the scouter's maximum number of moves, which is set according to the expected cover time and a safety function, before considering a corresponding node as not connected. The main idea is to remove from the *scouter* information of nodes – sorted by increasing order of their counter value, where the gap between successive nodes (k^{th} and $k-1^{th}$) is greater than the number of moves required to explore k connected elements in a random walk fashion. After having the *scouter* perform a large enough number of moves, the number of nodes in the system can be estimated by simply counting the number of elements kept in the set of visited nodes.

Other relevant approaches based on the execution of random walks to collect samples, like *Sample & Collide* [56; 98] and *Capture-Recapture* [94], are described in Section 3.2.5.

3.1.5 Gossip-based

Commonly, gossip and epidemic communication are indistinctly referred. However, in a relatively recent review of gossiping in distributed systems [82] a slight distinction

between the two is made. In a nutshell, the difference simply relies on the interaction directionality of both protocols. The authors state that gossiping is referred to "the probabilistic exchange of information between two members", and epidemic is referred to "information dissemination where a node randomly chooses another member". Even so, the effect of both protocols in terms of information spread is much alike, and strongly related to epidemics. Notice that, the information spread in a group in real life (gossip) is similar to the spread of an infectious disease (epidemics). For this reason, in this work no distinction will be made between gossip and epidemic protocols.

Gossip communication protocols are strongly related to epidemics, where an initial node ("infected") sends a message to a (random) subset of its neighbors ("contaminated"), which repeat this propagation process – "one to many". With the right parameters, almost the whole network will end up participating in this propagation scheme. This communication pattern exhibits interesting characteristics despite its simplicity, allowing a robust (fault tolerant) and scalable information dissemination over all the network, in a completely decentralized fashion. Nevertheless, it is important to point out that the robustness of gossip protocols may not be directly attained by any algorithm based on a simple application of this communication pattern. For instance, an algorithm correctness may rely on principles and invariants that may not be guaranteed by a straightforward and incautious use of a gossip communication protocol, as revealed in [71]. In general, gossip communication tends to be as efficient as flooding, in terms of speed and coverage, but it imposes a lower network traffic load (to disseminate data).

Push-Sum Protocol The push-sum protocol [80] is a simple gossip-based protocol to compute aggregation functions, such as SUM or AVERAGE, consisting of an iterative pairwise distribution of values throughout all the network. In more detail, along discrete times t, each node i maintains and propagates information of a pair of values (s_{ti}, w_{ti}): s_{ti} represents the sum of the exchanged values, and w_{ti} denotes the weight associated to this sum at the given time t and node i. In order to compute distinct aggregation functions, it is enough to assign appropriate initial values to these variables. E.g., considering v_i as the initial input value at node i, AVERAGE: $s_{0i} = v_i$ and $w_{0i} = 1$ for all nodes; SUM: $s_{0i} = v_i$ for all nodes, only one node starts with $w_{0i} = 1$ and the remaining assume $w_{0i} = 0$; COUNT: $s_{0i} = 1$ for all nodes, only one with $w_{0i} = 1$ and

the others with $w_{0i} = 0$.

Independently from the computed aggregation function, the algorithm proceeds in the following way. In each iteration, a neighbor is chosen uniformly at random, and half of the actual values are sent to the target node and the other half to the node itself. Upon receive, the local values are updated, adding each value from a received pair to its local component (i.e., pointwise sum of pairs). The estimate of the aggregation function can be computed by all nodes, at each time t by s_{ti}/w_{ti}. The accuracy of the produced result will tend to increase progressively along each iteration, converging to the correct value. As referred by the authors, the correctness of this algorithm relies on a fundamental property defined as the *mass conservation*, stating that: the global sum of all network values (local value of each node plus the value in messages in transit) must remain constant along time. Considering the crucial importance of this property, the authors assume the existence of a fault detection mechanism, that allow nodes to detect when a message did not reach its destination. In this situation, the "mass" is restored by sending the undelivered message to the node itself. This algorithm is further generalized by the authors in their work – *push-synopses protocol*, in order to combine it with random sampling to compute more "complex" aggregation functions (e.g., quantiles) in a distributed way.

Other approaches In the last years, several gossip-based approaches have been proposed, due to the attractive characteristics of gossip communication: simplicity, scalability and robustness. Several alternative algorithms inspired by the *push-sum protocol* have been proposed, like: *Push-Pull Gossiping* [67; 68] which provides an anti-entropy aggregation technique (see Section 3.2.2), or *G-GAP* [128] (Gossip-based Generic Aggregation Protocol) that extends the *push-synopses protocol* to tolerate non contiguous faults (i.e., neighbors can not fail within the same short time period).

Another aggregation algorithm supported by an information dissemination and group membership management protocol, called *newscast protocol*, is proposed in [66]. This approach consists of the dissemination of a cache of items (with a predefined size) maintained by each network node. Periodically, each node randomly selects a peer, considering the network addresses of nodes available on the local cache entries. The cache entries are exchanged between the two nodes and the received information is merged into their local cache. The merge operation discards the oldest items, keeping a predefined number of the freshest ones, also ensuring that there is at most one item

from each node in the cache. An estimate of the desired aggregate can be produced by each network node, by applying the aggregation function to the local cache of items.

3.1.6 Hybrid

Hybrid approaches combine the use of different communication techniques to obtain improved results from their synergy. Commonly, the use of a hierarchic topology is mixed with gossip communication. Hierarchic based schemes are efficient and accurate, but highly affected by the occurrence of faults. On the other hand, gossip based algorithms are more resilient to faults, but less efficient in terms of message load (requiring more message exchanges). In general, this combination enables hybrid approaches to achieve a fair trade-off between performance (in terms of message load and accuracy) and robustness, when performing aggregation in more realistic environments.

(Chitnis et al., 2008) Chitnis et al. [23] studied the problem of computing aggregates in large-scale sensor networks in the presence of faults, and analyzed the behavior of hierarchy-based (i.e., TAG) and gossip-based (i.e., Push-Sum Protocol) aggregation protocols. In particular, they observe that tree-based aggregation is very efficient for very small failures probabilities, but its performance drops rapidly with increasing failures. On the other hand, a gossip protocol is slightly slowed down (almost unaffected), and is better to use with failures (compared to tree-based). Considering these results, the authors proposed an hybrid protocol with the intent of leveraging the strengths of both analyzed mechanisms and minimize their weakness, in order to achieve a better performance in faulty large-scale sensor networks.

This hybrid approach divides the network nodes in groups, and a gossip-based aggregation is performed within each one. A leader is elected for each group, and an aggregation tree is constructed with the leader nodes (multi-hop routing may be required between leaders) to further perform a tree-based aggregation with the results from each gossip group. The authors also defined and solved an optimization problem to get the best combination between the two aggregation mechanisms, yielding the optimal size of the groups, according to the network size and failure probability. However, in practice this requires the pre-computation of the gossip group size (by solving the referred optimization problem) before starting to use of the protocol with optimal settings. Results from simulations show that the hybrid aggregation approach usually

outperforms the other two (tree-based and gossip-based) [2].

An extension of the previous approach for heterogeneous sensor networks is later discussed in [24]. In this case, it is considered that a few distinguished nodes, designated as *microservers*, which are more reliable and less prone to failure than the remaining ones, are available in the network. The aggregation technique works mostly like the one previously described for the homogeneous case, but with two differences that take advantage of the reliability of *microservers*. First, microservers are preferably chosen as group leaders. Second, microservers are put on the top of the created aggregation tree which may also be composed by other less reliable nodes. The use of microservers in the aggregation tree will increase its robustness, and by putting them at the top it will reduce the need to reconstruct the whole tree when a fault occurs. The evaluation results show that the aggregation process can be enhanced in heterogenous networks, when taking advantage of more reliable (although more expensive) nodes.

Other Approaches A more elaborated structure was previously defined by Astrolabe [125]. Astrolabe is a DNS-like distributed management system that supports attributes aggregation. It defines a hierarchy of zones (similar to the DNS domain hierarchy), each one holding a list of attributes called MIB (Management Information Base). This structure can be viewed as a tree, each level composed of non-overlapping zones, where leaf zones are single hosts, each one running an Astrolabe agent, and the root zone includes all the network. Each zone is uniquely identified by a name hierarchy (similarly to DNS), assigning to each zone a unique string name within the parent zone; the global unique name of each zone is obtained by concatenating the name of all its parent zones from the root with a predefined separator. The zone hierarchy is implicitly defined by the name administratively set to each agent. A gossip protocol is executed between a set of elected agents to maintain the existing zones. The MIB held by each zone is computed by a set of aggregation functions, that produce a summary of the attributes from the child zones. An aggregation function is defined by a SQL-like program that is code embedded in the MIB, being set as a special attribute. Agents keep a local copy of a subset of all MIBs, in particular of zones in the path to the root and siblings, providing replication of the aggregated information with weak consistency (eventual consistency). A gossip protocol is used for agents to exchange data about MIBs from other (sibling) zones and within its zone, and update its state

[2] Notice that only static network settings (no node arriving/leaving) were considered by the authors.

with the most recent data.

Another hierarchical gossiping algorithm was introduced by Gupta et. al [60], being one of the first to use gossip for the distributed computation of aggregation functions. According to the authors, the philosophy of this approach is similar to Astrolabe, but uses a more generic technique to construct the hierarchy, called *Grid Box Hierarchy*. The hierarchy is created by assigning (random or topology aware) unique addresses to all members, generated from a known hash function. The most significant digits of the address are used to divide nodes into different groups (grid boxes) and define the hierarchy. Each level of the hierarchy corresponds to a set of grid boxes, matching a different number of significant digits. The aggregation process is carried out from the bottom to the top of the hierarchy in consecutive gossip phases (for each level of the hierarchy). In each phase: members of the same grid box gossip their data, compute the resulting aggregate after a predefined number of rounds, and then move to the next phase. The algorithm terminates when nodes find themselves at the grid box at the top of the hierarchy (last phase). Note that, this approach does not rely on any leader election scheme to set group aggregators, in fact the authors argue the inadequacy of such mechanism in unreliable networks prone to message loss and node crashes.

Recently, an approach that combines a hierarchy based technique with random sampling was proposed in [22] to approximate aggregation functions in large WSN. In this approach, the amount of collect data is regulated by a sampling probability produced from the input accuracy (expressed by two parameters ε and δ, i.e., relative error less than ε with probability greater than $1 - \delta$) and the aggregation function (i.e., COUNT, SUM or AVERAGE), aiming at reducing the energy consumption to compute the aggregate. This algorithm considers that the sensing nodes are organized in clusters (according to their geographic location), and that cluster heads form a spanning tree rooted at the sink. Basically, the aggregation proceeds as following: first, the sink computes the sampling probability p (according to ε and δ) and transmits it along with the aggregation function to all cluster heads across the spanning tree; then, cluster heads broadcast p to their cluster and each node within independently decides to respond according to the received probability; samples are collected at each cluster head which computes a partial result; finally, the partial results are aggregated upward the tree (convergecast) until the sink is reached, and where the final (approximated) result is computed. This algorithm, referred by the authors as BSC (Bernoulli Sampling

on Clusters), mixes the application of a common hierarchy based aggregation technique such as *TAG* (see Section 3.1.1) between cluster heads, with a flooding/broadcast method like *Randomized Reports* (see Section 3.1.3) to sample the values at each cluster.

3.2 Computation

In terms of computational principles on which the existing aggregation algorithms are based, the following main categories (see Table 3.2) were identified: *Hierarchical*, *Averaging*, *Sketches (hash or min-k based)*, *Digests*, and *Sampling*. These categories intrinsically support the computation of different kinds of aggregation functions. For instance, *Hierarchical* approaches allow the computation of any decomposable function. *Averaging* techniques allow the computation of all duplicate sensitive decomposable functions that can be derived from the AVERAGE, by using specific initial input values and combining the results form different instances of the algorithms. *Sketches* techniques also allow the computation of duplicate sensitive decomposable functions, but that can be derived from the SUM [3]. Moreover, schemes based on hash sketches are natively able to compute distinct counts (non decomposable duplicate insensitive), and those based on min-k can be easily adapted to compute it (e.g., in extrema propagation, see 3.2.3, using the input value as seed of the random generation function, so that duplicated values will generate the same number). *Digests* support the computation of any kind of aggregation function, as this type of approach usually allows the estimation of the whole data distribution (i.e., values and frequencies) from which any function can be obtained. On the other hand, some techniques are restricted to the computation a single type of aggregation function, such as COUNT, which is the case of *Sampling* approaches.

Besides determining the supported aggregation function, the computational technique on which an aggregation algorithm is based constitutes a key element to define its behavior and performance, especially in terms of accuracy and reliability. *Hierarchical* approaches are accurate and efficient (in terms of message and computational complexity), but not fault tolerant. *Averaging* schemes are more reliable and also relatively accurate (converge along time to the correct result), although less efficient (requiring more message exchanges). Approaches based on the use of *sketches* are more reliable

[3] Note that, COUNT is the sum of all elements considering their input value as equal to 1.

than hierarchical schemes, adding some redundancy and providing fast multi-path data propagation, however they introduce an approximation error (depending on the number of inputs and size of the used sketch). *Digests* essentially consists on the reduction (compression) of all inputs into a fixed size data structure, using probabilistic methods and losing some information. Consequently, digests provide an approximation of the computed aggregation function, not the exact result. *Sampling* schemes are also based on probabilistic methods to compute the COUNT, being inaccurate and lightweight in terms of message complexity, as only a portion of the network is asked to participate.

In the following sections, the main principles and characteristics of these distinct classes are explained in a comprehensive way, and some important examples are described. A taxonomy of the identified computational principles is displayed in Table 3.2, associating them to the most relevant distributed aggregation algorithms.

3.2.1 Hierarchical

Hierarchical approaches take direct advantage of the self-decomposable property of some aggregation functions. Inputs are divided into separated groups and the computation is performed in a distributed way along a hierarchy (see Figure 2.1). Algorithms from this class depend on the previous creation of a hierarchic communication structure (e.g., tree, clusters hierarchy), where nodes can act as *forwarders* or *aggregators*. Forwarders simply transmit the received inputs to an upper level node. Aggregators apply the target aggregation function directly to all received input (and its own), and forward the result to an upper level node. The correct result is yield at the top of the hierarchy, being the aggregation process carried out from the bottom to the top of the hierarchy.

Algorithms from this class allow the computation of any decomposable function, providing the exact result (at a single node) if no faults occur. The global processing and memory resources required are equivalent to the ones used in a direct and centralized application of the aggregation function, but distributed across the network. However, these algorithms are not fault tolerant, e.g. a single point of failure may lead to the lost of all data beneath it.

Most of the algorithms from this category correspond to the ones belonging to the hierarchic communication class, like TAG [92], DAG [106], and I-LEAG [16]. Other algorithms can be found combining a hierarchical computation with another computation principle, namely: Tributary-Delta [95] mixes a hierarchical computation

Aggregation	Basis/Principles	Algorithms
Decomposable Functions	*Hierarchic*	TAG [92], DAG [106], I-LEAG [16], Tributary-Delta [95], (Chitnis et al., 2008) [23]
	Averaging	Push-Sum Protocol [80], Push-Pull Gossiping [67; 103; 68] DRG [21], (Chitnis et al., 2008) [23]
	Sketches	Sketches [29], RIA-LC/DC [49; 50] Extrema Propagation [8; 10] Tributary-Delta [95]
Complex Functions	*Digests*	Q-Digest[120], Equi-Depth[61], Adam2 [116]
Counting	*Sampling*	Random Tour [98], Randomized Reports [13], Sample & Collide [56; 98], Capture-Recapture [94], Hop-Sampling [83; 84], Interval Density [83; 84], (Kutylowski et al., 2002) [76] (Horowitz and Malkhi, 2003) [65],

Table 3.2: Taxonomy from a computation perspective.

with the use sketches in regions close to the *sink*; (Chitnis et al., 2008) [23] performs a hierarchic aggregation on the top of groups, and averaging is applied inside each one. See Sections 3.1.1 and 3.1.6 for more details about the aforementioned algorithms.

3.2.2 Averaging

The *Averaging* class essentially consists on the iterative computation of partial aggregates (averages), continuously averaging and exchanging data among all active nodes that will contribute to the obtention of the final result. This kind of approach tends to be able to reach a high accuracy, with all nodes converging to the correct result along the execution of the algorithm. A typical application of this method can be found in most gossip-based approaches (Section 3.1.5), where all nodes continuously distribute

a share of their value (averaged from received values) with some random neighbor, converging along time to the global network average (correct aggregation result). Algorithms from this category are more reliable than hierarchic approaches, working independently from the supporting network topology and producing the result at all nodes. However, they must respect an important principle, commonly designated as "mass conservation" in order to converge to the correct result. This invariant states that the sum of the aggregated values of all network nodes must remain constant along time [80].

Algorithms based on this technique are able to compute decomposable and duplicate-sensitive functions, which can be derived from the average operation; using different inputs initializations (e.g., COUNT), or combining functions executed concurrently (e.g., SUM, obtained by multiplying the results from an average and a count). In terms of computational complexity, this method usually involves the computation of simple arithmetic operations (i.e., addition and division), using few computational resources (processor and memory) and being fast to execute (at each node). This kind of algorithms are able to produce (almost) exact results, depending on their execution time (without failures). The minimum execution time required by these algorithms (number of iterations), to achieve a high accuracy, is influenced by the network characteristics (i.e., size, connection degree, and topology) and the communication pattern used to spread the partial averages. The robustness of this type of aggregation algorithm is strongly related with their ability to conserve the global "mass" of the system (see Chapter 4 for more details). Essentially, the loss of a partial aggregate ("mass") due to a node failure or a message loss introduces an error, resulting in the subtraction of the lost value from the initial global "mass" (leading to the non-contribution of the lost amount to the calculation of the final result, and therefore to the convergence to an incorrect value). In this kind of methodology, it is important to enforce the "mass" conservation principle, assuming itself as a main invariant to ensure the algorithms correctness.

Push-Pull Gossiping The push-pull gossiping [67] algorithm performs an averaging process, and it is gossip-based like the *push-sum protocol [80]* (previously described in Section 3.1.5). The main difference of this scheme relies on the execution of an anti-entropy aggregation process. The concept of anti-entropy in epidemic algorithms consists in the regular random selection of another site, to resolve all the differences

between the two, exchanging complete databases [34]. In particular, this algorithm executes an epidemic protocol to perform a pairwise exchange of aggregated values between neighbor nodes. Periodically, each node randomly chooses a neighbor to send its current value, and waits for the response with the value of the neighbor. Then, it averages the sent and received value, and calculates the new estimated value. Each time a node receives a value from a neighbor, it sends back its current one and computes the new estimate (average), using the received and sent values as parameters. In order to be adaptive and handle network changes (nodes joining/leaving), the authors consider the extension of the algorithm with a restarting mechanism (executing the protocol during a predefined number of cycles, depending on the desired accuracy). However, they do not address the "mass" conservation problem – impact of message losses or node failures.

A further study of this aggregation algorithm is discussed in [68], proposing a more mature solution that covers some practical issues: split the algorithm execution in two distinct threads; use of timeouts to detect possible faults, ignoring data exchanges in those situations; suggest different versions of the algorithm according to the aggregation function to compute; suggest the execution of several instances of the algorithm in parallel to increase its robustness.

DRG (Distributed Random Grouping) This approach [21] essentially consists on the continuous random creation of groups across the network, in which aggregates are successively computed (averaged). DRG was designed to take advantage of the broadcast nature of wireless transmission, where all nodes within radio range will be prone to ear a transmission, directing its application to WSN. The algorithm defines three different working modes for each node: *leader*, *member*, and *idle* mode. According to the defined modes and the performed state transitions, the execution of the algorithm can be separated in three main steps. First, each node in idle mode independently decides to become a group leader (according to a predefined probability), and consequently broadcast a GCM (Group Call Message) to all its neighbors, subsequently waiting for members. Second, all nodes in idle mode which received a GCM from a leader respond to the first one with a JACK (Joining Acknowledgment) tagged with their aggregated value, becoming members of that group (updating their state mode accordingly). In the third step, after gathering the group members values from received JACKs, the leader computes (averages) the group aggregate and broadcast a GAM (Group Assignment

Message) with the result, returning to idle mode. Each group member waits until it receives the resulting group aggregate from the leader to update its local value (with the one assigned in the GAM) and returns to idle mode, not responding to any other request until then.

The repeated execution of this scheme – creation of distributed random groups to perform in-group aggregation – allows the convergence of the estimate produced at all nodes to the correct aggregation result, as long as the groups overlap along time. The performance of DRG is influenced by the predefined probability of a node becoming leader, which determines its capacity to create groups (quantity and size of groups). Note that, in order to account for the occurrence of faults and avoid consequent deadlock situations that could arise in this algorithm, it is necessary to consider the definition of some timeouts (for the leaders to wait for JACKs, and the members to wait for a GAM). Intuitively, one will notice that the values set for these timeouts will highly influence the performance of the algorithm, although this detail is not addressed by the authors. An analysis of DRG on WSN with randomly changing graphs (modeling network dynamism) is provided in [20], assuming that the graph only changes at the beginning of each iteration of the algorithm (unrealistic assumption in practice, otherwise this leads to mass loss).

Other Approaches A well-known averaging approach, the Push-Sum (push-synopses) Protocol [80] has already been described in Section 3.1.5. In the last years, other approaches inspired by the Push-Sum Protocol have been proposed, intending to be more efficient in term of performance and robustness. Kashyap et al. [79] reduces the number of messages needed (communication overhead) to compute an aggregation function at the cost of an increase in the number of rounds. G-GAP (Gossip-based Generic Aggregation Protocol) [128] extends the *push-synopses protocol* [80] to support discontinuous failures (no adjacent node can fail within a period of 2 rounds) by restoring the mass loss resulting from failures (temporarily storing at each node previous data contributions).

Dimakis et al.[37; 36] propose an algorithm to improve the convergence time in random geometric networks. This scheme is similar to push-pull gossiping [67], differing on the peer selection methods. Instead of selecting a one-hop node as target of the averaging step, peers are selected according to their geographical location. In particular, a location is randomly chosen and the node closer to that local is selected.

A greedy geographic routing process is used to reach the node at the target location, assuming that nodes known their own geographic location.

Two averaging algorithms for asynchronous and dynamic networks are proposed in [99]. The core of the proposed schemes is based on a pairwise update, similarly to the push-pull gossiping (although not referred to the authors), addressing practical concerns that arise in asynchronous settings. In the first proposed algorithm nodes implement a blocking scheme to avoid the interference of other nodes in the update step and guarantee mass conservation. Additionally, a deadlock avoidance mechanism is considered, by imposing a sender-receiver relation on each link based on node UIDs. An extension to the first algorithm is proposed to cope with churn. The blocking mechanism (maintaining the directed relationship between nodes) is removed, and an additional variable is used to account for changes of each neighbor. When a node leaves the network, all its neighbors subtract the value associated to it from their state.

3.2.3 Sketches

The main principle of this kind of aggregation algorithm is based on the use of an auxiliary data structure with a fixed size, holding a *sketch* of all network values. The input values are used to create sketches that are aggregated across the network, using specific operations to update and merge them. Operations on sketches are order and duplicate insensitive, enabling them to be aggregated through multiple paths, being independent from the routing topology. This kind of technique is based on the application of a probabilistic method, generally allowing the estimation of the sum of the values held in the sketch.

Sketching techniques can be based on different methods, with different accuracy bounds and computational complexities. Algorithms from this class are mostly based on the application (with some improvements) of two main ideas: *hash sketches* [52; 127; 43; 53] and *k-mins sketches* [26].

Hash sketches allow the probabilistic counting of the number of distinct elements in a multiset (cardinality of the support set). This type of sketch essentially consists in a map of bits, initially set to zero, where each item is mapped into a position in the binary valued map (generally involving a uniform hashing function) setting that bit to one. The distinct count is estimated by checking the position of the most significant one bit (leftmost), or counting the number of zero bits in the sketch. The first hash sketching technique was proposed by Flajolet and Martin [52], being commonly des-

ignated as FM sketches (uniformly hash items into an integer, and maps only the less significant one bit of its bitmap representation to the sketch). In this first study, the authors also proposed the PCSA (Probabilistic Counting with Stochastic Averaging) algorithm to reduce the variance of the produced estimate, using multiple sketches and averaging their estimate (distributing the hash of an element to only one of the sketches). Another approach, Linear Counting [127] uses a hash function to directly map each element into a position of the sketch (setting that bit to one), and use the count of the number of zeros to produce an estimate. A further improvement to PCSA, designated LogLog, was described in [43], reducing required memory resources (an optimized version super-LogLog is also proposed, improving accuracy and optimizing memory usage applying a truncation and restriction rule). HyperLogLog [53] recently improved LogLog, consuming less memory to achieve a matching accuracy.

The k-mins sketches method was first introduced to determine the size of the transitive closure [4] in directed graphs [26]. It consists on assigning k independent random ranks to each item according to a distribution that depends on its weight, and keeping in a vector of the minimum ranks in the set. The obtained k-vector of the minimum ranks is used by an estimator to produce an approximated result. In other words, it can be said that k-mins sketches reduces the estimation of the sum to the determination of minimums of a collection of random numbers (generated using the sum operands as input parameters of the random distribution from which they are drawn). An improved alternative to k-mins sketches, designated bottom-k sketches, was recently proposed in [28].

The computational cost of sketching is dependent on the complexity of the operations involved in the creation and update of the sketches (e.g., hashing functions, random number generation, minimum/maximum determination), and the resources used by the estimator to produce a result. Algorithms based on sketches are not accurate, being based on probabilistic methods and introducing an error factor in the computed aggregation function. There is a trade-off between the accuracy and the size of the sketches. The greater the sketch size the tighter are the accuracy bounds of the obtained result, although requiring additional memory resources and a larger processing time. This kind of aggregation algorithm tends to be fast, although conditioned by the dissemination protocol used to propagate the sketches, being able to produce an approximate result after a number of iterations close to the minimum theoretical bound

[4] size of the union of the reachability sets (i.e., set of nodes that can be reached by a node, even through multiple hops) of all nodes.

(the network diameter).

RIA-LC/DC Fan and Chen [49] proposed a multi-path routing aggregation approach for WSN based on the use of LC (Linear Counting) sketches [127], which they later named RIA-LC (Robust In-network Aggregation using LC-sketches) [50]. The algorithm proceeds in two phases, like common multipath hierarchy-based approaches (see Section 3.1.1). In the first phase, the aggregation request (query) is spread from the sink throughout the whole network, creating a multipath routing hierarchy. In the second phase, starting at the lower level of the hierarchy, nodes respond to the aggregation request by creating a LC-sketch correspondent to its current local readings and sending it to the nodes at the upper level. All received sketches are combined with the local one (using the OR operation), and the result is sent to the next level until the top of the hierarchy is reached where the sink computes the aggregation estimate from the resulting LC-sketch.

Equation 3.1 is used to estimate the number of distinct items represented in a LC-sketch, where m is the size of the allocated bit vector, and z is the count of the number of bits with value equal to zero. In order to allow the computation of the SUM, each node creates a sketch by mapping a number of distinct items corresponding to its input value. For example, assuming that each node has a unique ID, if the node i has an input equal to 3, it maps the items $(ID_i, 1)$, $(ID_i, 2)$, and $(ID_i, 3)$ into the LC-sketch. In more detail, in this case the use of an hash function from the original LC-sketch design (to map duplicated items to the same bit) is replaced by a uniform random generator (since there are no duplicate items), randomly setting to 1 a number of bits equal to the input value.

$$\hat{n} = -m \ln (z/m) \tag{3.1}$$

The authors show by theoretical comparison and experimental evaluation that their approach outperforms the ones based on FM sketches [52], namely Sketches [29] (see 3.1.1), in terms of space and time requirements. They also claim a higher accuracy and lower variance when compared with existing sketch schemes. Moreover, they tackle some practical issues, like message size constraints, avoid the use of hash functions, and enable the specification of an approximation error.

Recently, the authors improved RIA-LC by considering the use of sketches with variable sizes instead of fixed size sketches, referring to the new technique as RIA-

DC (Robust In-network Aggregation using Dynamic Counting sketches) [50]. The authors observed that the large preallocated sketches used in RIA-LC were wasting space, since at the beginning of the computation must of the bits are set to zero. In RIA-DC the initial size of sketches is variable and depends on the local sensor reading. Along the aggregation process the size of the sketches is adjusted (gradually increasing toward the sink), in order to satisfy a given accuracy constrain. RIA-DC decreases message overhead and energy consumption compared to RIA-LC, keeping similar accuracy properties.

Extrema Propagation This approach reduces the computation of an aggregation function, more precisely the sum of positive real numbers, to the determination of the minimum (or maximum) of a collection of random numbers [8; 10]. Initially, a vector x_i of k random number is created at each network node i. Random numbers are generated according to a known random distribution (e.g., exponential or gaussian), using the node initial value v_i as the input parameter for the random generation function (e.g., as the rate of an exponential distribution). Then, the execution of the aggregation algorithm simply consists of the computation of the pointwise minimum (or alternatively maximum) between all exchanged vectors. This technique supports the use of any information spreading algorithm as a subroutine to propagate the vectors, since the calculation of minimums is order and duplicate insensitive. In particular, the authors consider that at each round all nodes send their resulting vector to all their neighbors.

At each node, the obtained vector is used as a sample to produce an approximation of the aggregation function, applying a maximum likelihood estimator derived from extreme value theory (branch of statistics dealing with the extreme deviation from the median of a probabilistic distribution). For example, considering the generation at each node of k random numbers with an exponential distribution of rate v_i, and the use of the minimum function to aggregate the vectors. Equation 3.2 gives the estimator for the SUM of all v_i from the sample of minimums $x_i[1], ...x_i[k]$ in the vector x_i, with variance $\text{SUM}^2/(k-2)$:

$$\widehat{\text{SUM}} = \frac{k-1}{\sum_{j=1}^{k} x_i[j]} \tag{3.2}$$

This algorithm is focused on obtaining a fast estimate, rater than an accurate one. Although, the accuracy of this aggregation algorithm can be improved by using vectors of larger size, adjusting k to the desired relative accuracy (e.g., $k = 387$ for a maximum

relative error of 10%, with a confidence of 95%). A further extension to the protocol to allow the determination of the network diameter has been proposed in [18].

Other Approaches A representative approach based on FM sketches has already been described in Section 3.1.1 – Sketches [29]. In this multi-path approach, a generalization of PCSA is used to distinguish the same aggregates received from multiple paths, and subsequently manage to compute duplicate-sensitive aggregation functions. Other similar approaches can be found in the literature based on hash sketches, like *Synopsis Diffusion* [108] and Wildfire [14]. These approaches apply essentially the same aggregation process, operating in two phases (request/response) and only differing on small aspects.

Synopsis Diffusion [108] is an aggregation approach for WSN close to the one proposed by Sketches [29]. In a sense, this work presents a more generic framework relying on the use of duplicate insensitive summaries (i.e., hash sketches), which they call ODI (Order- and Duplicate-Insensitive) synopses. Namely, they generically define the synopses functions (i.e., generation, fusion and evaluation) required to compute aggregation functions, and provide examples of ODI synopses to compute more "complex" aggregates (i.e., not decomposable aggregation functions). For instance, besides the scheme based on FM sketches, they propose other data structures (and respective functions) to uniformly sample sensor readings and compute other sampling based aggregation functions. The authors also tackled additional practical concerns. Namely, they explored the possibility of implicitly acknowledging ODI synopses to infer messages losses, and suggested simple heuristics to modify the established routing topology (assigning nodes to another hierarchic level), in order to reduce loss rate.

Wildfire [14] is based on the use of FM sketches to estimate SUM, but it is targeted for dynamic networks. Despite the fact of operating in two phases like previous hash sketch approaches, unlike them it does not establish any specific routing structure (i.e., multipath hierarchy) to aggregate sketches. After receiving the query, nodes start combining the received sketches with their current one, and then send the result if it differs from the previous one.

A distributed implementation of some basic hash sketches schemes has been proposed in [109; 110]. DHS (Distributed Hash Sketches) is supported by a DHT, taking advantage of the load balancing properties and scalability of such structure. More specifically, the authors describe how to build DHS based on PCSA [52] and supper-

LogLog [43].

Mosk-Aoyama and Shah [105; 104] proposed an algorithm, called COMP, to compute the sum of values from individual functions (referred to as separable functions). This algorithm is very similar to *Extrema Propagation* but less generic, as it is restricted to the properties of exponential random variables distribution. Furthermore, COMP uses a biased estimator, being less accurate than Extrema Propagation which uses unbiased ones.

3.2.4 Digests

This category includes algorithms that allow the computation of more complex aggregation functions, like quantiles (e.g., median) and frequency distributions (e.g., mode), besides common aggregation functions (e.g., count, average and sum). Basically, algorithms from this class produce a *digest* that summarizes the system data distribution (e.g., histogram). The resulting *digest* is then used to approximate the desired aggregation functions. We refer to a *digest* as a data structure with a bounded size, that holds an approximation of the statistical distribution of input values in the whole network. This data structure commonly corresponds to a set of values or ranges with an associated counter.

Digests provide a fair approximation of the data distribution, not holding an exact representation of all the system values due to efficiency and scalability reasons. The accuracy of the result obtained from a digest depends on its quality (i.e., used data representation) and size. Digest allow the computation of a wider range of aggregation functions, but usually require more resources and are less accurate than the other more specialized approaches.

Q-Digest An aggregation scheme that allow the approximation of complex aggregation functions in WSN is proposed in [120]. This approach is based on the construction and dissemination of q-digests (quantile digests) along a hierarchical routing topology (without routing loops and duplicated messages). A q-digest consists of a set of buckets, hierarchically organized, and their corresponding count (frequency of the values contained by the bucket). Buckets are defined by a range of values $[a, b]$ and can have different sizes, depending on the distribution of values they represent. Each node maintains a q-digest of the data available to it (from its children). Q-digests are built in a bottom-up fashion, by merging received digests from child nodes, and fur-

ther compressing the resulting q-digest according to a specific compression factor (less frequent values are grouped in large buckets). Aggregation functions are computed by manipulating (e.g., sort q-digest nodes) and traversing the q-digest structure according to a specific criteria (depending on the function to be computed).

The authors provide an experimental evaluation, where they show that q-digests allow the approximation of quantile queries using fixed message sizes, saving bandwidth and power when compared to a naive scheme that collects all the data. The naive scheme obtains an exact result, but with increasing message size along the routing hierarchy. Obviously, there is a trade-off between the obtained accuracy and the message size used. The authors suggest a way to compute the confidence factor associated to a q-digest (i.e., error associated to a query), but the effect of faults is not considered in their study.

Equi-Depth A gossip-based approach to estimate the network distribution of values is described in [61]. This scheme is based on the execution of a gossip protocol and the application of specific merge functions to the exchanged data, to restrict storage and communication costs. In more detail, each node keeps a list of k values (digest), initially set to its input value. At each round, nodes get the list of values from a randomly chosen neighbor and merge it with their own, applying a specific procedure. The result from the execution of several rounds produces an approximation of the network distribution of values (i.e., histogram). Four merging techniques were considered and analyzed by the authors: *swap*, *concise counting*, *equi-width histograms*, and *equi-depth histograms*.

Swap simply consists in randomly picking k values from the two lists (half from each of them) and discarding the rest. Although simpler, by discarding half of the available data in each merge, important information is likely to be lost.

Concise counting associates a tuple, value and count, to each list entry. The merge process consists in sorting the tuples (by value), and individually merging the tuples with the closest values, in order to keep a fixed list size. Tuples are merged by randomly choosing one of the values and adding their count.

The equi-width technique breaks the range of possible values into bins of equal size, associating a counter to each one. Initially, nodes consider the range from 0 to the current input value, as the extremes are not known. Bins are dynamically resized when new extremes are found: all bins are mapped into larger ones, based on their

middle value and the range of the new bin, adding their counter to the new mapped bin. This technique requires only the storage of the extreme values and counts, since all bins have an equal width, reducing the volume of data that needs to be stored and exchanged when compared to other techniques (e.g., concise counting). However, equi-width can provide very inaccurate results for severely skewed distributions.

In equi-depth, bins are divided not to be of the same width but to contain approximately the same count. Initially, fixed size bins are set, each represented by a pair <value, counter>, dividing the range from 0 to the input value. Whenever data is exchanged, all pairs (received and own) are ordered, and consecutive bins that yield the smallest combined bins (in terms of count) are merged, repeating the process until the desired number of bins is obtained. Bin merge consists in adding the counters and using the arithmetic weighted mean as value. This method intends to minimize the counting disparity across bins.

In order to deal with changes in input values along time, the authors consider the execution of the protocol in phases, restarting it. The authors experimentally evaluated their protocol comparing the previous merging techniques. The results obtained show that equi-depth outperformed the other approaches, providing a consistent trade-off between accuracy and storage requirements for all tested distributions. The author also evaluated the effect of duplicates, from the execution of the gossip protocol. They argue from the results obtained that although duplicates bias the estimated result, it is more advantageous (simpler and efficient) to assume their presence than to try to remove them. The occurrence of faults and change in the input values were not evaluated.

Adam2 Adam2 is a gossip based algorithm to estimate the statistical distribution of values across a decentralized system [116]. More precisely, this scheme approximates the CDF (Cumulative Distribution Functions) of an attribute, which can then be used to derive other aggregates. In this case, a "digest" is composed by a set H_i of k pairs of values (x_k, f_k), where x_k represents an interpolation point and f_k is the fraction of nodes with value less or equal than x_k. At a high abstraction level, it can be said that the algorithm simply executes several instances of an averaging protocol (i.e., Push-Pull Gossiping [68]) to estimate the fraction of nodes in each pair of the CDF.

In more detail, each node can decide to start an instance of Adam2 according to a predefined probability $\frac{1}{\hat{n}_i R}$, where $\hat{n}_i$ is the current network size estimate at node i

and R is an input parameter that regulates the aggregation instances frequency (i.e., on average one every R rounds). Each instance is uniquely identified by its starting node. Initially, the starting node i initializes the interpolation set H_i in the following way: fractions f_k are set to 1 if the node attribute reading v_i is less or equal than the corresponding interpolation value x_k, and set to 0 otherwise. Nodes store a set of interpolation points H_i for each running algorithm instance (initiated by a node i). Upon learning about a new instance, a node j initializes its H_i setting $f_k = 1$ if $a_j \leq x_k$ and $f_k = 0$ otherwise, and starts participating in the protocol. A push-pull like aggregation is then performed, where nodes randomly choose a neighbor to exchange their set H_i, which are subsequently merged by averaging the fractions at each interpolation point. Along time, the fractions will converge at each node to the correct result associated to each pair. After a predefined number of rounds (*time-to-live*) the CDF is approximated by interpolating the points of the resulting set H_i. Note that, Adam2 concurrently estimates (by averaging) other aggregation functions besides CDF, namely COUNT to determine the network size, and MIN/MAX to find the extreme attribute values. The result from these aggregation functions are later used as input values to the next instances of the algorithm to tune and optimize its execution (i.e., calculate the instance starting probability, and set new interpolation points).

Like in Push-Pull Gossiping [67; 68], Adam2 handles dynamism (i.e., attribute changes and churn) by continuously starting new instances of the algorithm – restart mechanism. The authors evaluated the algorithm by simulation, comparing it with previous techniques to compute complex aggregates (e.g., *Equi-Depth*). The results obtained show that Adam2 outperforms the compared approaches, exhibiting better accuracy.

Other Approaches One of the first algorithms to compute complex aggregation functions in WSN was introduced by Greenwald and Khanna [59]. Their approach is similar to the one previously described for q-digest (3.2.4): nodes compute quantile summaries (digest) that are merged in a bottom-up fashion along a tree topology, until the root is reached.

Another gossip based scheme to estimate the distribution of input readings, able to detect outliers, was introduced in [47; 48]. In a nutshell, this approach operates like the push-sum protocol [80] (described in Section 3.1.5), but manipulates a set of clusters (digests) instead of a single value, applying a specific clustering procedure.

In general, existing aggregation approaches can be extended to compute more complex aggregation functions, for instance combining them with an additional sampling technique. However, this additional functionality is not part of the essence of their core algorithm, bearing different characteristics (e.g., accuracy) and concerns. Some examples can be found in [80] where push-sum is extended with a push-random protocol to obtain random samples, and in [27] which introduces algorithms to estimate several spatially-decaying aggregation functions.

3.2.5 Counting

This category refers to a restricted set of distributed algorithms, designed to compute a specific aggregation function: COUNT. COUNT allows the determination of important properties in the design of some distributed applications. For instance, in this context it finds a common practical application in the determination of the size of the system (or group), or to count the number of votes in an election process. The algorithms from this class rely on the use of some randomized process, most of them usually based on the execution of some *sampling* technique to provide a probabilistic approximation of the size of the sample population. Nonetheless, a few algorithms are found that do not explicitly collect samples for size estimation, instead applying a probabilistic *estimator* over some observed events.

Algorithms based on sampling are strongly influenced by the probabilistic method used to obtain the result, inheriting its properties. For instance, the accuracy of the algorithm corresponds to the one provided by the probabilistic method used, being bounded by the error factor associated with it. Several probabilistic methods have been applied to samples to yield a counting estimation, namely: *birthday problem* [31] – concerns the probability of two elements sampled out of a population not being repeated, inspired from the probability of two people out of a group not having a matching birthday; *capture-recapture* [118] – probabilistic method based on the repeated capture of samples from a closed population (population that maintains a fixed size during the sampling process), where the number of common elements between samples are accounted to provide an estimate of the population size; *fundamental probabilistic methods* – application of Bernoulli based sampling methods [22], and other basic probabilistic concepts on some sampled statistical information, like the distances between nodes (number of hops) or the number of messages successfully sent/received, in order to estimate de size of the network. In all cases, typically sampling is performed at a

single node, and it can take several rounds to collect a single sample. Moreover, an estimation error is always present, even if no faults occur. For example, in *Sample & Collide* [56; 98] the estimation error can reach 20%, and a sampling step takes $\bar{d}T$ (where $\bar{d}$ is the average connection degree and T is a predefined timer that must be sufficiently large to provide a good sample quality), needing to be repeated until l sample collisions are observed.

As previously referred, in some cases a size estimation can be obtained by directly applying an estimator on some available system knowledge (observed events or other known properties), without any previous explicit sampling. Although, in general the estimator inputs result from other sampling sources. For instance, in the approach proposed by Horowitz and Malkhi [65] (see Section 3.1.2) an estimator function is used at each node to estimate the network size, based on the observation of two events (nodes joining or leaving the network), incrementing/decrementing the estimator. In this case the nodes joining/leaving at each node can be seen as the input sample used to provide the estimate. Other approaches, like the one proposed in [39; 40], provide a size estimation based on knowledge of the routing structure, in this particular case counting the number of high degree nodes (which can be considered the input sample). This kind of techniques does not provide accurate results, in most cases yielding a rough approximation to the correct value.

Sample & Collide This approach [56; 98] addresses the problem of counting the number of peers in a P2P overlay network, inspired by a birthday problem technique (first proposed by Bawa et al. on a technical report [13]). The application of this probabilistic method requires the collection of uniform random samples. To this end, the authors proposed a peer sampling algorithm based on the execution of a continuous time random walk, in order to obtain unbiased samples (asymptotically uniform). The sampling routine proceeds in the following way: an initiator node i sets a timer with a predefined value T, which is sent in a sampling message to a randomly selected neighbor; upon receiving a sampling message, the target node (or the initiator after setting the timer) picks a random number U uniformly distributed within the interval $[0, 1]$, and decrements the timer by $\log(1/U)/d_i$ (i.e., $T \leftarrow T - \log(1/U)/d_i$, where d_i is the degree of node i); if the resulting value is less or equal than zero ($T \leq 0$) then the node is sampled, its identification is returned to the initiator and the process stops; otherwise the sampling message is sent to one of its neighbors, chosen uniformly

at random. The quality of the samples obtained (approximation to a uniform random sampling) depends on the value T initially set to the timer. The described sampling step (to sample one peer) must be repeated until one of the nodes is repeatedly sampled a predefined number of times l (i.e., l sample collisions are observed). After concluding this sampling process, the network size n is estimated using a Maximum Likelihood (ML) method. The ML estimate can be computed by solving Equation 3.3, where C_l corresponds to the total number of samples until one is repeated l times, using a standard bisection search. Alternatively, the result can be approximated within $\sqrt{n}$ of the ML-estimator by Equation 3.4 (asymptotically unbiased estimator), which is computationally more efficient.

$$\sum_{i=0}^{C_l-l-1} \frac{i}{n-1} - l = 0 \tag{3.3}$$

$$\widehat{n} = C_l^2/2l \tag{3.4}$$

The accuracy of the produced result is determined by the parameter l, and its fidelity depends on the capacity of the sampling method to provide uniformly distributed random samples (T must be sufficiently large).

Capture-Recapture Mane et al. [94] proposed an approach based on the capture-recapture statistical method to estimate the size of closed P2P networks (i.e., networks of fixed size, with no peers joining or leaving during the process). This method requires two or more independent random samples from the analyzed population, and further counting the number of individuals that appear repeated in each sample. The authors use random walks to obtain independent random samples. Considering a two-sample strategy, two random walks are performed from a source node, one in each sampling phase (capture and recapture). In more detail, each random walk proceeds in the following way: the source node sends a message to a randomly selected neighbor, which at its turn forwards the message to another randomly chosen neighbor; the process is repeated until a predefined maximum number of hops is reached (parameter: *time-to-live*) or the message gets back to a node that has already participated in the current random walk. During this process, the information about the traversed path (i.e., the UIDs of all participating nodes) is kept in the forwarded message. When one of the random walk stopping criteria is met, the message is sent back to the source node with the list of the "captured" nodes, following the reverse traversed path (stored

in the message). The information received at the source node from the sampling steps is used to compute the estimate $\widehat{n}$ of the network size, applying Equation 3.5 (where n_1 is the number of nodes caught in the first sample, n_2 is the number of nodes caught in the second sample, and n_{12} represent the number of recaptured nodes, i.e. caught in both samples).

$$\widehat{n} = \frac{((n_1 + 1) \times (n_2 + 1))}{(n_{12} + 1)} \tag{3.5}$$

Hop-Sampling One of the approaches proposed by Kostoulas et al. [83; 84] to estimate the size of dynamic groups is based on sampling the receipt times (hop counts) of some nodes from an initiator. Receipt times are obtained across the group from a gossip propagation started by a single node, the initiator, that will further sample the resulting hop counts of some nodes to produce an estimate of the group size. In more detail, the protocol proceeds as following: the initiator starts the process by sending an *initiating* message (to itself); upon receiving the initiating message nodes start participating in the protocol, forwarding it to a number (*gossipTo*) of other targets, until a predefined number of rounds (*gossipFor*) is exceeded, or a maximum quantity of messages (*gossipUntil*) have been received; gossip targets are chosen uniformly at random from the available membership, excluding nodes in a locally maintained list (*fromList*) from which a message has already been received; exchanged messages carry the distance to the initiator node, which is measure in number of hops; each node keeps the received minimum number of hops (*MyHopCount*), and sends the current value incremented by one. After concluding the described gossip process, waiting for a predefined number of rounds (*gossipResult*), the initiator samples the number of hops (*MyHopCount*) from some nodes selected uniformly at random. The average of the sampled hop counts is then used to estimate the logarithm of the size of the group ($log(n)$). Alternatively to the previous sampling process, where nodes wait for the initiator sample request, nodes can decide themselves to send their hop count value back to the initiator node, according to a predefined probability to allow only a reduced fraction of nodes to respond.

Interval Density A second approach to estimate the size of a dynamic group has been proposed by Kostoulas et al. in [83; 84]. This algorithm measures the density of the process identifiers space, determining the number of unique identifiers within

a subinterval of this space. The initiator node passively collects information about existing identifiers, snooping the information of complementary protocols running on the network. The node identifiers are mapped to a point in the real interval $[0, 1]$ by applying a hash function to each one. The initiator estimates the group size by determining the number of sampled identifiers X lying in a subinterval I of $[0, 1]$, returning X/I. Notice that this kind of approach assumes a uniformly random distribution of the identifiers, or uses strategies to reduce the existing correlation between them, in order to avoid biased estimations.

Other Approaches Some counting approaches based on a centralized probabilistic polling to collect samples were previously described in this work (in Section 3.1.3). Namely, *randomized reports* which illustrate the basic idea of probabilistic polling, and another approach [76] that samples the number of message successfully sent in a single-hop wireless network (further improved in [78]).

Other probabilistic polling algorithms are also available in the specific context of multicast groups, to estimate their membership size. For example, in [54] some older mechanisms were analyzed and extended, and in [4] an algorithm using an estimator based on Kalman filter theory was proposed to estimate the size of dynamic multicast groups.

Chapter 4

Dependability Issues of Existing Algorithms

The previous chapter presented an overview of the existing related work, succinctly describing the most relevant distributed aggregation algorithms. In this chapter, the dependability issues found on existing approaches are discussed, especially when confronted with message loss and churn. As we will see, most of the current distributed aggregation techniques are not fault-tolerant. In particular, the occurrence of faults can drastically affect the accuracy of the produced result. The few approaches that tolerate faults are unable to produce precise results. Moreover, few are able to operate in dynamic settings, in general relying on a restart mechanism to handle churn (periodically resetting the computation), and revealing several flaws. A particular attention will be given to averaging approaches (gossip based), due to their appealing robustness characteristics.

Single tree based approaches can be drastically and unpredictably affected by a single failure in the aggregation topology, since the failure of a single node lead to the (temporary) disconnection of all its subtree. Experimental results [92] have shown that in realistic WSN scenarios, without using any technique to counteract failures, depending on the network diameter, the percentage of nodes involved in the data aggregation process is reduced, and can range from less than 10% to about 40%. Even using some techniques to reduce the effect of message loss, such as caching schemes, the quantity of nodes not involved in the process can come up to 30%, unpredictably affecting the result accuracy depending on the relevance of the lost data. Interpolation strategies based on past values can be used to replace unavailable values due to mes-

sage loss, such as caching the previous data of children, as suggested in [92]. However, despite providing an overall improvement on the obtained accuracy in the presence of losses, the use of caching schemes bring on some drawbacks, like: temporal smearing of individual sensor readings; use of additional memory (that could be needed for other purposes); the cache validity duration sets the minimum bound to detect node departure. Furthermore, tree based approaches are completely dependent on the topology maintenance and recovery scheme to work in dynamic and mobile environments. The efficiency of the topology maintenance protocol will directly influence the performance of the aggregation algorithm in dynamic settings. Moreover, beside consuming additional resources to monitor the network, in order to detect changes, the topology adaptation process (parent switching) can still cause temporary disconnection that can affect the aggregation process. For all these reasons, no guarantees on the quality of the computed result can be made using this kind of aggregation technique, as a single loss can jeopardize the final result (loosing input data).

Multi-path based approaches usually aim at preventing the problem of single point of failure from the common tree/cluster based approaches, establishing alternative paths from each node to the *sink*. However, this strategy gives rise to the duplication of data, since the same value can be transmitted through different paths. Algorithms supported by this kind of routing topology must be able to handle duplicated data, to correctly compute duplicate-sensitive aggregation functions. Commonly, these approaches use an auxiliary duplicate insensitive structure to summarize all values, and apply an estimation function to that structure to produce the result (e.g., sketches [29]). This kind of technique introduces an approximation error in the final result, even if no fault occurs. More importantly, the use of sketches is not tailored to operate in dynamic settings, as in the case of departures (even when announced) it is not trivial (if not impossible) to remove items from such structures [81].

The use of sketches removes the dependency from a specific routing topology to perform data aggregation, enabling algorithms to benefit from the existing path redundancy to increase their resilience. However, this comes at the price of losing accuracy, even if no fault occurs, due to the application of probabilistic estimators to compute the final result. This issue is observed in all *randomized* approaches, like sampling. In the case of algorithms based on random walks, despite being able to operate on unstructured networks, a singe failure (e.g., message loss) can interrupt the protocol, losing the token that executes the random walk.

Different from other approaches, that are also independent from the routing topology, averaging techniques are supposedly accurate (converging along time to the correct result). This type of algorithm is intended to be robust, being able to produce an estimate of the aggregation function at every node. In particular, these algorithms are often based on a gossip (or epidemic) communication scheme, which is commonly thought to be robust. Nevertheless, similarly to gossip communication protocols [5], the robustness of such aggregation algorithms can be challenged, according to the assumptions made on the environment in which they operate. In practice, averaging techniques, exhibit relevant problems that have been overlooked, not supporting message loss nor node crashes. Due to their importance, the next section exposes relevant dependability issues of popular algorithms from this class (see [71] for more details).

4.1 Robustness of Averaging Algorithms

The correctness of averaging-based aggregation algorithms depends on the maintenance of a fundamental invariant, commonly designated as "mass conservation". This property states that the sum of the aggregated values of all network nodes must remain constant along the algorithm's execution, in order for it to converge to the correct result [80]. However, considering realistic settings (e.g., message loss) this invariant if often broken, making the algorithms converge to a wrong value. To illustrate the issues of the existing averaging approaches, three representative algorithms are discussed: Push-Sum Protocol [80], Push-Pull Gossiping [67; 68], and Distributed Random Grouping [21].

4.1.1 Push-Sum Protocol

The Push-Sum Protocol (PSP) [80] is a simple gossip-based aggregation algorithm, essentially consisting on the distribution of shares across the network. Each node maintains and iteratively propagates information of a *sum* and a *weight*, which are sent to randomly selected nodes (more details in Section 3.1.5).

Aware of the impact of "mass" loss, the authors considered a variation of the algorithm to cope with message loss: all nodes must possess the ability to detect when their messages did not reach their destination, and send the undelivered data to the node itself, in order to recover the lost mass. But, this is an unrealistic assumption. Using an acknowledgement-based scheme to infer message loss, as suggested, would amount to

solving the *coordinated attack problem*, which under possible message loss has been shown to be impossible [58]. Furthermore, even if it was possible, it would introduce additional waiting delays in the protocol, in order to receive a delivery notification for each sent message.

PSP does not support node crash and departure. In order to be robust against node failures, G-GAP [128] extended the PSP, implementing a scheme based on the computation of recovery shares and the explicit acknowledgement of mass exchanges between peers. However, this approach provides only a partial support against this type of faults, supporting discontiguous node crashes (it assumes that no two adjacent nodes fail within a short time interval apart).

A recent extension to PSP has been proposed in [81] to address its use in dynamic environments. An additional step is added to the original algorithm, by simply replacing a portion λ of the current node mass with an equivalent portion of its initial value, at each iteration. This strategy introduces an error, according to the used λ, that adjusts the aggregated values toward the nodes initial value, and improves the estimate produced when node crashes or leave the network. However this technique does not solve the mass loss problem.

4.1.2 Push-Pull Gossiping

The Push-Pull Gossiping (PPG) [67; 68] is based on an anti-entropy aggregation process, being quite similar to PSP (more details in Section 3.2.2). The main difference of this algorithm relies on its *push-pull* process, which enforces a symmetric pairwise mass exchange between peers. This action/reaction pattern between two nodes aims to promptly nullify their differences. The iterative execution of this push-pull process across all the network will provide the convergence of the algorithm to the correct value (faster, when compared to PSP).

The authors implicitly assume that the core of the push-pull process is atomic, also referred as "the variance reduction step" ($v_i = v_j = (v_i + v_j)/2$). However, this atomicity constraint is not met even in a fault-free synchronous model, where message sending takes one round (making a push-pull span two rounds), and makes it give incorrect results. In particular, a node that started a push-pull can have its state updated before receiving the pull value to average, violating the mass conservation invariant and consequently the correctness of the algorithm, as depicted in Figure 4.1. In this case, node b updates its state $(2 + 1)/2 = 1.5$ before completing an already

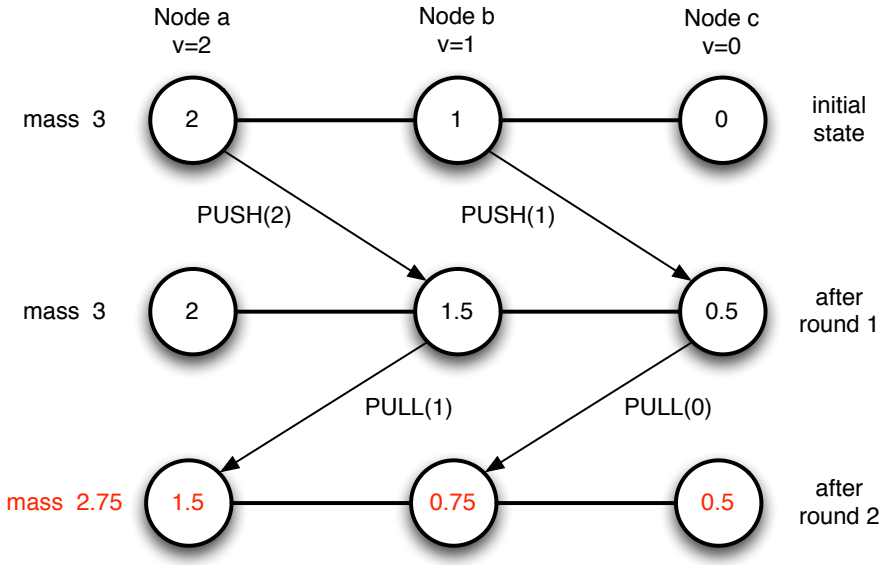


Figure 4.1: Violation of the mass conservation invariant in the Push-Pull Gossiping protocol (estimates at the end of each round).

started push-pull, leading to a state update $(1.5 + 0)/2 = 0.75$ after receiving the pull message from node c that is inconsistent with the value previously sent, when both should have updated their aggregate to $(1 + 0)/2 = 0.5$.

In practice, additional modifications must be considered to guarantee a behavior equivalent to an atomic *push-pull*. Two such improvements to the algorithm (*back cancellation* and *ordered wait*) were proposed in [71]. The proposed fixes solve the convergence problem of PPG, but it comes at the cost of a performance degradation (especially, convergence speed), being *Push Pull Ordered Wait* the one with the best performance. In particular, this fix prevents nodes that already started a push-pull from participating in another one, buffering a received push message and only replying to it after receiving and processing the pull message they are waiting for. This blocking mechanism ensures a consistent state update between participating nodes, guaranteeing the atomicity of the push-pull process. To avoid deadlock (i.e., nodes waiting for each other), a push is only allowed to nodes with a greater UID.

Considering faults, the loss of (pull) messages and node crashes originate a violation of the mass conservation property. The consequent occurrence of estimation errors is recognized by the authors of PPG [68], but no effective strategy that solves this issue is proposed (only statistically reducing the impact of the error).

The authors also propose a periodic restart of the PPG algorithm to cope with

churn, reinitializing the execution of the algorithm with clean input values (restoring the correct mass of the system) after a predefined number of rounds – *epoch*. New nodes are only able to participate in the protocol at the next epoch, for the algorithm to converge to the average of the values at the start of each epoch. This strategy limits the accuracy of the algorithm and its response to network changes: if the epoch length is too small, it will react quickly to change but provide less accurate results; on the other hand, if the epoch is too long it will react slowly to changes. In Section 7.5, it is shown that the use of a restart mechanism, to handle churn, introduces a delay in the response to network changes.

4.1.3 Distributed Random Grouping

Distributed Random Grouping (DRG) [21] was designed to take advantage of the broadcast nature of wireless transmissions (where all nodes within radio range will be prone to hear a transmission), directing its use for WSN (unlike previous averaging algorithms). In a nutshell, the algorithm essentially consists of the continuous creation of random groups across the network, to successively perform in-group aggregations. Over time, ensuring that the created groups overlap, the estimated values at all nodes will converge to the correct network-wide aggregation result (see Section 3.2.2 for more details).

Although not addressed by the authors, in practice since messages can be lost, timeouts are needed in the reception steps for expected JACK and GAM messages, to exempt nodes (leaders and members respectively) from waiting forever. The values of those timeouts will influence the performance of the algorithm.

In particular, the authors consider the occurrence of collisions and link failures, but they only consider its effect on GMCs, with impacts on the creation of groups, reducing its expected size. For instance, they assume that link failures only happen between iterations of the protocol, which is unrealistic since links can unpredictably fail at any point of the algorithm execution, causing the loss of any type of message. The loss of a GCM will have no impact on the correctness of the algorithm, only preventing nodes from joining the group. However, loosing a JACK from a node but delivering the subsequent GAM to that same node will violate mass conservation. The same happens if GAMs are lost.

In line with the previous approaches, this algorithm does not support node crashes and departures. Recently, an analysis of DRG considering dynamic graph changes was

provided in [20]. However, the authors assumed that changes only occur at the beginning of an iteration of the algorithm, which once more is unrealistic, and overlooked its effect at the middle of an iteration (namely at the JACK/GAM level, incurring from the previously referred mass conservation problem). To cope with the change of the sensor measurements over time, the authors consider the execution of several instances of DRG (associated to a respective timestamp).

Part III

Robust Distributed Aggregation Approach

Chapter 5

Flow Updating

This chapter describes the main contribution of this research work, a robust distributed aggregation approach named *Flow Updating* [72; 74]. Considering the taxonomy proposed in Chapter 3, it can be classified as an averaging algorithm in terms of computational principles, working independently from the routing topology (i.e., unstructured group) and fitting in the gossip category in term of communication. For starters, the basis behind this novel approach and an intuition of how the algorithm works will be given. Then, the algorithm is described in more detail in Section 5.1. A more profound analysis and a proof of the algorithm correctness is provided in Section 5.2. Finally, some possible variations (optimizations) to the algorithm are introduced in Section 5.3, for instance adapting it to operate on asynchronous settings.

Flow Updating (FU) is inspired from the concept of network flows, from graph theory (which serves as an abstraction for many things like water flow or electric current; see Chapter 6 of [35]). It is rooted in current averaging approaches, which iteratively average the values between peers, converging at all nodes to global system wide average (see Section 3.2.2). Classic schemes start from the initial input value (to average) and iteratively change it by exchanging "mass" along the execution of the algorithm. Unlike them, FU keeps the initial input unchanged, exchanging and updating flows associated to neighbors. The key idea is to explore the concept of *flow*, and instead of storing the current estimate (i.e., average) at each node in a variable, compute it from the input value and the contribution of the flows along the edges to the neighbors:

$$e_i = v_i - \sum_{j \in \mathcal{D}_i} f_{ij}. \tag{5.1}$$

The above expression can be read as: the current estimate e_i in a node i is the input value v_i less the flows f_{ij} from the node to each neighbor j, where $\mathcal{D}_i$ represents the set of neighbors (i.e adjacent nodes) of i.

In a sense, flows represent the value that must be transferred between two adjacent nodes for them to produce the same estimate, and are skew symmetric (i.e., the flow value from i to j is the opposite from j to i: $f_{ij} = -f_{ji}$). For example: considering two directly connected nodes i and j with initial input values $v_i = 1$ and $v_j = 3$, for them to produce the same average $\frac{1+3}{2} = 2$, the flows at node i and j must be respectively set to $f_{ij} = -1$ and $f_{ji} = 1$.

In a nutshell, the core of the algorithm is the following: each node i stores the flow f_{ij} to each neighbor j; node i sends flow f_{ij} and its estimate e_i to j in a message; a node j receiving f_{ij} updates its own f_{ji} with $-f_{ij}$, and afterwards uses e_i to compute its new estimate e_j (by averaging), setting new flows accordingly. Messages lead to simple flow updates, being idempotent; the value in a subsequent message overwrites the previous one, it does not add to the previous value. If the skew symmetry of flows holds, the sum of the estimates for all nodes (the global mass) will remain constant, where $\mathcal{V}$ represents the set of all network nodes:

$$\sum_{i \in \mathcal{V}} e_i = \sum_{i \in \mathcal{V}} (v_i - \sum_{j \in \mathcal{D}_i} f_{ij}) = \sum_{i \in \mathcal{V}} v_i \tag{5.2}$$

The intuition is that if a message is lost the skew symmetry is temporarily broken, but as long as a subsequent message arrives, it re-establishes the symmetry. The reality is somehow more complex: due to concurrent execution, messages between two nodes along a link may cross each other and both nodes may update their flows concurrently; therefore, the symmetry may not hold, but what happens is that along time $f_{ij} + f_{ji}$ converges to 0, and the global mass converges to the sum of the input values of all nodes. Message loss only delays convergence, it does not impact the correct value (see Section 5.2.2.1).

Enforcing the skew symmetry of flows along edges through idempotent messages is what confers Flow Updating its unique fault tolerance characteristics, that distinguish it from previous approaches. It tolerates message loss by design without requiring additional mechanisms to detect and recover mass from lost messages. It solves the mass conservation problem, observed in current *averaging* approaches (see Section 4.1). Next, the main version of the algorithm is described in more detail.

inputs:
  v_i, value to aggregate
  $\mathcal{D}_i$, set of neighbors given by failure detector
state variables:
  flows: initially, $F_i = \{\}$
message-generation function:
  $\text{msg}_i(F_i, j) = (i, f, \text{est}(v_i, F_i))$;
  with $f = \begin{cases} F_i(j) & \text{if } (j, \_) \in F_i \\ 0 & \text{otherwise} \end{cases}$
state-transition function:
  $\text{trans}_i(F_i, M_i) = F_i'$
  with
  $F = \{j \mapsto -f \mid j \in \mathcal{D}_i \wedge (j, f, \_) \in M_i\} \cup$
  $\{j \mapsto f \mid j \in \mathcal{D}_i \wedge (j, \_, \_) \notin M_i \wedge (j, f) \in F_i\}$
  $E = \{i \mapsto \text{est}(v_i, F)\} \cup$
  $\{j \mapsto e \mid j \in \mathcal{D}_i \wedge (j, \_, e) \in M_i\} \cup$
  $\{j \mapsto \text{est}(v_i, F_i) \mid j \in \mathcal{D}_i \wedge (j, \_, \_) \notin M_i\}$
  $a = (\sum\{e \mid (\_, e) \in E\})/\,|E|$
  $F_i' = \{j \mapsto f + a - E(j) \mid (j, f) \in F\}$
estimation function:
  $\text{est}(v, F) = v - \sum\{f \mid (\_, f) \in F\}$

Algorithm 1: Flow Updating algorithm.

5.1 Algorithm

Flow Updating is described considering the execution of the algorithm under the synchronous network model (as in Chapter 2 of [90]). More specifically, the computation proceeds in lockstep rounds, each one composed by two steps: first nodes look at their state and compute what messages are sent, through a *message-generation function*; then nodes take their state and the messages received and compute a new state, through a *state-transition function*. It is assumed that each node needs only to be able to distinguish its neighbors, not requiring the use of globally unique identifiers. According to this model, the computation performed at each node by FU is depicted by Algorithm 1.

The algorithm takes two *inputs* that can be read at each round, but are not updated

by the algorithm itself (line 2–3). The first one, v_i, is the input value of the aggregation functions that will be computed. The second, $\mathcal{D}_i$ represents the set of neighbors of node i. Considering the execution of the algorithm in dynamic settings, these values can change. But, as we will see later on, a simple action (or none) in response to the observed change will be enough to allow the algorithm to cope with it, without interrupting (e.g., restarting) its execution. We will get back to this topic at the end of this section.

The state of each node i simply consists of a mapping F_i from node IDs to flows, that stores for each current neighbor the flow along the edge toward that node (line 5). At each round, each node can estimate the computed aggregate (i.e., average) by applying the estimation function (line 20) to the current input value v_i and flows F_i, according to Equation 5.1, i.e., $e_i = \text{est}(v_i, F_i)$.

A single type of message is sent, containing the self ID i of the node, the flow $F_i(j)$ to each current neighbor j, and the aggregate estimate (lines 7–8). When no flow value is available for a given neighbor, initially or when a new node starts participating, the value 0 is used. The estimate is computed by making use of the *estimation function* (line 20). Note that it is expected that the message is multicast to all neighbors, ideally in a single transmission when supported by the physical communication medium. In practice, this can be easily achieved in wireless networks taking advantage of the shared communication medium and the radio broadcast found in such environments, or in other network settings by considering the creation of multicast groups composed by a node and its neighbors.

The state-transition function (lines 10–18) takes a state F_i and the set of messages M_i received by the node in the round, and returns a new state F_i'. Basically, it is here that the averaging step is performed, taking the received estimates and the current one to produce the new average. Then, a new set of flows (i.e., the new state) is computed, in order for each neighbor to produce the new average. More specifically, some auxiliary variables are used in Algorithm 1 to compute the new state: the flows F updated according to the messages received, and the estimates E (last received) used to calculate the new average a. Looking at these values in more detail:

- F is a mapping from current neighbor IDs to: the symmetric of the flow in messages, for those successfully received from neighbors (line 12); the current value, if any, in the case of message loss (line 13).

- E is a mapping from node IDs to estimates: for the self node i, according to the

estimation function, using the newly updated flows in F (line 14); for neighbors whose messages arrived, the estimate received (line 15); otherwise, for neighbors from which no message was received, the estimate according to the estimation function, using the flows at the beginning of the round, i.e. the estimate sent to all neighbors at the beginning of the round (line 16).

- a is simply the average of the estimates in the mapping E, i.e. the averaging step, and represents the new estimate towards which the node will lead its neighbors to converge in the next round.

Finally, the new state, i.e. the mapping F_i', is computed by adjusting each flow in F, adding the difference between the new average a and the last (received) estimate $E(j)$, so that the estimates change toward a. In particular, the estimate of node i at the end of the round will be a, and its neighbors would also compute a at the end of the next round if they do not receive other messages from their own neighbors (e.g., if it has node i as its only neighbor).

The iterative execution of this algorithm across the whole network allows the estimate of all nodes to converge to the global average of the input values. The occurrence of message loss will not change the value to which the algorithm will converge (only delaying it), unlike in previous averaging approaches. FU is also able to support churn, simply by performing straightforward operations on the map of flows F: removing the entries corresponding to leaving (or crashing[1]) nodes, and adding flow entries for newly arriving nodes. The map F is maintained according to the set of alive neighbors $\mathcal{D}_i$ which is updated by a Fault Detector (FD). Notice that practical implementations of FDs can be used. An evaluation of the use of different FD is available in Section 7.5.1.1. The dynamic change of the input value v_i over time (e.g., temperature) is seamlessly supported by Flow Updating, without requiring any additional action, converging at each iteration to the new global average of input values (Section 7.5.2).

In short, FU is able to continuously adapt to changes (i.e., churn and input value changes) without any kind of restart mechanism (unlike the remaining distributed aggregation approaches), being fault tolerant. Like other averaging based aggregation approaches, FU allows the computation of several aggregation functions by combining the use of different input values with the execution of distinct instances of the algorithm. For example: AVERAGE is computed by default; COUNT is calculated using

[1]Nodes departure is considered equivalent to node crash, because it is assumed that nodes silently leave the network (without any kind of notification, since it can be lost).

1 as the input value in c nodes (usually one, i.e. $c = 1$) and $v_i = 0$ at the remaining nodes, estimating the result at each node by c/e_i; SUM can be computed by combining the result from the execution of two instances of FU, one to determine the average $\hat{a}$ and another the number of elements $\hat{n}$ (i.e., count), returning the sum as $\hat{a}\hat{n}$.

5.2 Correctness

After previously describing how FU works, its correctness is analyzed in this section, providing a proof of its convergence along time, according to a concurrent model close to the one considered in the previous section, and discussing the impact of message loss on the correctness of the algorithm. As already referred previously, in terms of correctness, exiting averaging algorithms rely on the maintenance of a fundamental invariant, commonly designated as "mass conservation". This invariant states that the global mass of the system remains constant along time.

Definition 5.1 (Global Mass)**.** *The "global mass" M of the network is the sum of the aggregate value a_i (i.e., the estimated average) held by all nodes:*

$$M = \sum_{i \in \mathcal{V}} a_i$$

(considering that the system is stopped, with no messages in transit and the current round successfully completed at every node).

FU distinguishes itself from the existing averaging algorithms by its fault-tolerant capabilities. It solves the mass conservation problem observed on other averaging approaches when subject to message loss, that affect their correctness leading them to converge to a wrong value. In particular, other approaches require additional mechanism to detect and restore the lost mass, which is not feasible in practice. In contrast, FU is by design able to support message loss, only delaying the convergence to the correct value, without requiring any additional mechanism. This is achieved by keeping the input values unchanged and performing idempotent flow updates which, together with the skew symmetric property lead to conservation of the global mass.

This claim will be substantiated along this section. For that purpose, different models will be considered, from simpler to more sophisticated, aiming at reaching a correctness proof for a version as close as possible to the fully concurrent algorithm previously described.

In fact, despite the efforts made, we were unable to prove the convergence of the algorithm considering a fully concurrent execution, as expressed by Algorithm 1, where all nodes execute the algorithm at the same time, messages cross each other, and flows are concurrently updated on the same link. To this end, the execution of FU was modeled as a discrete dynamical system, i.e., a difference equation, but the stability analysis was inconclusive. Even resorting to the help of other researchers specialized in the area (i.e., dynamic systems, matrix analysis, and linear algebra), no advances were achieved. Even though, a characterization of FU as a discrete dynamic system can be found in Appendix A, which can be used as a rigorous tool to predict its behavior in specific network settings, and be used as a starting point to carry out further correctness analysis in a future work.

Nevertheless, a correctness proof is still given considering a concurrent execution of FU, although imposing a restriction to avoid concurrent flow updates over any single link (see Section 5.2.3). For ease of understanding, first the correctness of FU is analyzed in a non concurrent model (see Section 5.2.2), and only then considering a concurrent execution of the algorithm at all nodes.

5.2.1 Model and Assumptions

In this section, we describe a general model and list some assumptions common to all analyzed versions of the algorithm. First, we consider a discrete time execution of the algorithm (in lock-step, $t \geq 0$ and $t \in \mathbb{N}$) over a network, that is abstracted by an undirected graph $G(\mathcal{V}, \mathcal{E})$. The set of vertices $\mathcal{V}$ represent the network nodes, and the edges $\mathcal{E}$ the existing links between nodes which are all bidirectional. The graph is connected, meaning that there is a path between any two vertices of G, and of fixed size n, with no vertices or edge modifications along time, where $n = |\mathcal{V}|$. Unless otherwise stated, no faults occur along the execution of the algorithm (i.e., no node fail/crash), and the communication channels are assumed to be reliable (i.e., no duplicate, spurious or lost messages). It is assumed that each node i knows a set of neighbors $\mathcal{D}_i$, and only communicates with them. All nodes are identical processes, not requiring a global UID, only a local relative identification to distinguish neighbors. The size of the set of neighbors of a node i is denoted by $|\mathcal{D}_i|$, which corresponds to the node *degree*. No further assumptions are made on the network, and its size n is assumed to be unknown.

Along this analysis, we consider the problem of the distributed computation of the

AVERAGE aggregation function, i.e., determination of the global average $\bar{a}$ of all input values v_i held by each node i in a distributed network system.

According to the above considerations, in the following sections we gradually define some properties and analyze the correctness of our approach, considering different models and versions of our algorithm with increased approximation to the fully concurrent original algorithm, but maintaining the core idea of flow exchange. Next, the correctness analysis of Flow Updating is carried out considering these assumptions, unless stated otherwise.

5.2.2 (Simplest) Non Concurrent Model

First, a restrictive (not realistic) but very simple model is considered to start the correctness analysis of FU. In this model, only one node executes the algorithm at each round, and it is assumed that all nodes have the same opportunity to execute the algorithm along time (e.g., uniformly chosen at random), maintaining all the assumptions previously defined (Section 5.2.1).

A simplified version of the FU algorithm is considered (see Algorithm 2), with the same core as the one previously described in 5.1, but here message exchanges are abstracted. Comparing to Algorithm 1, we should notice the following differences and similarities: the inputs v_i and $\mathcal{D}_i$ are the same, but do not change along the execution of the algorithm; the state also consists in a simple map of flows F_i, but it is initialized with the value zero for all neighbors, as the network is fixed (no nodes leaving/arriving) and no entry will be removed/added to F_i; the message generation function was removed, as message exchange is abstracted, and updates are directly performed on the state of the target nodes; the use of the auxiliary flow F and estimate E to hold default values for unreceived messages was removed, since no message is lost; the estimation function is identical, but only applied to state flows; in the averaging step, the new estimate (i.e., average a_i) also results from calculating the average of all neighbors estimates and the one of the executing node obtained from the flows of its neighbors (i.e., e'_i); finally, the new flows toward each neighbor are computed in the same way, adding the difference between the new and previous estimate to the previous (current) flow, and each corresponding flow at each neighbor is updated with the skew symmetric value.

Starting the analysis of FU, according to this model, the first property that can be observed is that at the end of each round of execution of the algorithm, all the

1 **inputs:**
2 v_i, value to average
3 $\mathcal{D}_i$, set of known neighbors
4 **state variables:**
5 flows: initially, $F_i = \{j \leftarrow 0 \mid j \in \mathcal{D}_i\}$
6 **algorithm:**
7 $e'_i = v_i + \sum_{j \in \mathcal{D}_i} F_j(i)$
8 $a_i = \frac{\sum_{j \in \mathcal{D}_i} e_j + e'_i}{|\mathcal{D}_i| + 1}$
9 **forall the** $j \in \mathcal{D}_i$ **do**
10 $F_i(j) = F_i(j) + (a_i - e_j)$
11 $F_j(i) = -F_i(j)$
12 **estimation function:**
13 $e_i = v_i - \sum_{j \in \mathcal{D}_i} F_i(j)$

Algorithm 2: Simplest Flow Updating algorithm, abstracting message exchange.

flows between neighbors (i.e., adjacent nodes) are skew symmetric, as stated by the following lemma.

Lemma 5.1 (Skew Symmetry)**.** *At the end of each round, for all $i \in \mathcal{V}$:*

$$F_i(j) = -F_j(i), \forall_{j \in \mathcal{D}_i}$$

Proof. Initially all flows are set to zero, then at round $t = 0$, $\forall_{i \in \mathcal{V}}$, $F_i(j)^t = -F_j(i)^t$, $\forall_{j \in \mathcal{D}_i}$ holds trivially. As only one node executes the algorithm at each round, only the flows of that node and those of its neighbors are updated; no concurrent state updates are performed by any other nodes. Considering that at any round $t > 0$ a single node i executes the algorithm, from line 9-11 of the Algorithm 2 we directly observe that $\forall_{j \in \mathcal{D}_i}$ each neighbor flow $F_j(i)$ is updated with the opposite (skew symmetric) value of its corresponding flow at node i, i.e. $-F_i(j)$ (line 11). □

Consequently, the global mass of the system is conserved at each iteration, as the sum of all flows is zero.

Lemma 5.2 (Mass Conservation)**.** *At the end of each round:*

$$M = \sum_{i \in \mathcal{V}} e_i = \sum_{i \in \mathcal{V}} v_i$$

Proof. Let $M = \sum_{i \in \mathcal{V}} v_i$. Now, considering the sum of the estimates at all nodes, i.e. $\sum_{i \in \mathcal{V}} e_i$, and substituting e_i by the estimation function of Algorithm 2 (line 13), we get:

$$\begin{aligned} \sum_{i \in \mathcal{V}} e_i &= \sum_{i \in \mathcal{V}} \left(v_i - \sum_{j \in \mathcal{D}_i} F_i(j) \right) \\ &= \sum_{i \in \mathcal{V}} v_i - \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{D}_i} F_i(j) \end{aligned}$$

From Lemma 5.1, we infer that the sum of the flows along any edge is equal to zero, $F_i(j) - F_j(i) = 0$, and consequently the sum of all flows in the network is also zero. Therefore $\sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{D}_i} F_i(j) = 0$ for all rounds and then $\sum_{i \in \mathcal{V}} e_i = \sum_{i \in \mathcal{V}} v_i = M$. □

Another important property is that, after concluding the execution of the algorithm at a node i, all its neighbors and itself will yield a common estimate which corresponds to the newly computed average. In a sense, this characterizes the averaging process performed by FU, where at one iteration the values of the involved participants (node i and its neighbors) are averaged, and all will return the resulting average as their estimate. This property is expressed by the following lemma.

Lemma 5.3 (Flow Adjustment)**.** *If i executes Algorithm 2 at round $t \geq 0$, then at the end of that round:*

$$e_i^t = e_j^t = a_i^t, \forall_{j \in \mathcal{D}_i}$$

Proof. First, let's show that the estimate of the node i chosen to execute the algorithm at round t is equal to the new computed average (i.e., $e_i^t = a_i^t$) at the end of the round. We know that the estimate is given by the estimation function (line 13 of Algorithm 2), i.e. $e_i^t = v_i - \sum_{j \in \mathcal{D}_i} F_i(j)^t$, and that the flow values to all neighbors of node i will result

from $F_i(j)^t = F_i(j)^{t-1} + (a_i^t - e_j^{t-1})$ (line 10 of Algorithm 2), which means that:

$$\begin{aligned} e_i^t &= v_i - \sum_{j \in \mathcal{D}_i} \left(F_i(j)^{t-1} + (a_i^t - e_j^{t-1}) \right) \\ &= v_i - \sum_{j \in \mathcal{D}_i} F_i(j)^{t-1} - |\mathcal{D}_i|\, a_i^t + \sum_{j \in \mathcal{D}_i} e_j^{t-1} \\ &= e_i^{t-1} - |\mathcal{D}_i|\, a_i^t + \sum_{j \in \mathcal{D}_i} e_j^{t-1} \end{aligned}$$

Then, from the average computation, line 8 of Algorithm 2, and given that $e_i^{t'} = e_i^{t-1}$ attending to the skew symmetry of flows (Lemma 5.1), we have $(|\mathcal{D}_i| + 1)a_i^t = \sum_{j \in \mathcal{D}_i} e_j^{t-1} + e_i^{t-1}$. Substituting the sum of estimates in the above expression, we get:

$$\begin{aligned} e_i^t &= (|\mathcal{D}_i| + 1)a_i^t - |\mathcal{D}_i|\, a_i^t \\ &= a_i^t \end{aligned}$$

Now, let's show that the estimate of any neighbor j of the node i that executes the algorithm is also equal to the new computed average (i.e., $e_j^t = a_i^t$) at the end of the round. Concerning each neighbor j, we observe that only their flow associated to i is updated by Algorithm 2 (line 11). Therefore, at the end of round t, the estimate of any neighbor j results from subtracting to its initial value v_j the flows of its neighbors k (that were unchanged) and the flow set by i. More specifically, the estimate of each neighbor j is given by (evidencing the flow associated to i):

$$e_j^t = v_j - \sum_{k \in \mathcal{D}_j \setminus \{i\}} F_j(k)^t - F_j(i)^t$$

From lemma 5.1 $-F_j(i)^t = F_i(j)^t$, and as $F_i(j)^t = F_i(j)^{t-1} + (a_i^t - e_j^{t-1})$ (line 10 of Algorithm 2):

$$e_j^t = v_j - \sum_{k \in \mathcal{D}_j \setminus \{i\}} F_j(k)^t + F_i(j)^{t-1} + a_i^t - e_j^{t-1}$$

Substituting e_j^{t-1} by the corresponding estimation function at round $t-1$, and evidencing the flow associated to i:

$$e_j^t = v_j - \sum_{k \in \mathcal{D}_j \setminus \{i\}} F_j(k)^t + F_i(j)^{t-1} + a_i^t - v_j + \sum_{k \in \mathcal{D}_j \setminus \{i\}} F_j(k)^{t-1} + F_j(i)^{t-1}$$

Since only flows associated to node i are updated, we get:

$$e_j^t = F_i(j)^{t-1} + a_i^t + F_j(i)^{t-1}$$

Finally, again from lemma 5.1 $F_i(j)^{t-1} = -F_j(i)^{t-1}$, we obtain:

$$\begin{aligned} e_j^t &= -F_j(i)^{t-1} + a_i^t + F_j(i)^{t-1} \\ &= a_i^t \end{aligned}$$

□

Now, attending to the above properties it is possible to show that there is a variance reduction at each round, and the estimates of all nodes will converge along time to the global network average $\bar{a}$ (i.e., $\bar{a} = \sum_{i \in \mathcal{V}} v_i$), by executing the algorithm. Let θ_t^2 be the variance of the estimated values across all the network at round t, which is given by:

$$\theta_t^2 = \frac{1}{n-1} \sum_{i \in \mathcal{V}} (e_i^t - \bar{a})^2 \tag{5.3}$$

Lemma 5.4 (Variance Reduction)**.** *At the end of round t:*

$$\theta_t^2 \leq \theta_{t-1}^2$$

Proof. We will show that $\theta_t^2 - \theta_{t-1}^2 \leq 0$:

$$\theta_t^2 - \theta_{t-1}^2 = \frac{1}{n-1} \left(\sum_{v \in \mathcal{V}} (e_v^t - \bar{a})^2 - \sum_{v \in \mathcal{V}} (e_v^{t-1} - \bar{a})^2 \right)$$

Let $\mathcal{P}$ be the set of participating nodes involved in the execution of the algorithm (i and its neighbors $\mathcal{D}_i$), i.e. $\mathcal{P} = \mathcal{D}_i \cup \{i\}$, and $\mathcal{R}$ the set of remaining nodes that do not participate in the algorithm at round t, i.e. $\mathcal{R} = \mathcal{V} \backslash \mathcal{P}$. Rewriting the previous expression to separate these two sets, we get:

$$\theta_t^2 - \theta_{t-1}^2 = \frac{1}{n-1}\left(\sum_{p\in\mathcal{P}}(e_p^t - \bar{a})^2 + \sum_{r\in\mathcal{R}}(e_r^t - \bar{a})^2 - \sum_{p\in\mathcal{P}}(e_p^{t-1} - \bar{a})^2 - \sum_{r\in\mathcal{R}}(e_r^{t-1} - \bar{a})^2\right)$$

Since the estimates of the set of non participating nodes are unchanged, i.e. $\forall_{r\in\mathcal{R}}, e_r^t = e_r^{t-1}$, the reasoning can be narrowed to the set of participating nodes $\mathcal{P}$:

$$\begin{aligned}
\theta_t^2 - \theta_{t-1}^2 &= \frac{1}{n-1}\left(\sum_{p\in\mathcal{P}}(e_p^t - \bar{a})^2 - \sum_{p\in\mathcal{P}}(e_p^{t-1} - \bar{a})^2\right) \\
&= \frac{1}{n-1}\left(\sum_{p\in\mathcal{P}}\big((e_p^t)^2 + \bar{a}^2 - 2e_p^t\bar{a}\big) - \sum_{p\in\mathcal{P}}\big((e_p^{t-1})^2 + \bar{a}^2 - 2e_p^{t-1}\bar{a}\big)\right) \\
&= \frac{1}{n-1}\left(\sum_{p\in\mathcal{P}}\big((e_p^t)^2 - 2e_p^t\bar{a}\big) - \sum_{p\in\mathcal{P}}\big((e_p^{t-1})^2 - 2e_p^{t-1}\bar{a}\big)\right)
\end{aligned}$$

From Lemma 5.3, the estimates of all participants at the end of the round will be equal, i.e. $\forall_{p\in\mathcal{P}}, e_p^t = a_i^t$, getting:

$$\begin{aligned}
\theta_t^2 - \theta_{t-1}^2 &= \frac{1}{n-1}\left(\sum_{p\in\mathcal{P}}\big((a_i^t)^2 - 2a_i^t\bar{a}\big) - \sum_{p\in\mathcal{P}}\big((e_p^{t-1})^2 - 2e_p^{t-1}\bar{a}\big)\right) \\
&= \frac{1}{n-1}\left(|\mathcal{P}|\,(a_i^t)^2 - |\mathcal{P}|\,2\bar{a}a_i^t - \sum_{p\in\mathcal{P}}(e_p^{t-1})^2 + 2\bar{a}\sum_{p\in\mathcal{P}}e_p^{t-1}\right)
\end{aligned}$$

Now, from the average computation (line 8 of Algorithm 2) and as $e_i^{t'} = e_i^{t-1}$ due to the skew symmetry of flows (Lemma 5.1), replacing a_i^t by the average formula, we obtain:

$$\begin{aligned}
\theta_t^2 - \theta_{t-1}^2 &= \frac{1}{n-1}\left(|\mathcal{P}|\left(\frac{\sum_{p\in\mathcal{P}}e_p^{t-1}}{|\mathcal{P}|}\right)^2 - |\mathcal{P}|\,2\bar{a}\left(\frac{\sum_{p\in\mathcal{P}}e_p^{t-1}}{|\mathcal{P}|}\right) - \sum_{p\in\mathcal{P}}(e_p^{t-1})^2 + 2\bar{a}\sum_{p\in\mathcal{P}}e_p^{t-1}\right) \\
&= \frac{1}{(n-1)}\left(\frac{1}{|\mathcal{P}|}\left(\sum_{p\in\mathcal{P}}e_p^{t-1}\right)^2 - 2\bar{a}\sum_{p\in\mathcal{P}}e_p^{t-1} - \sum_{p\in\mathcal{P}}(e_p^{t-1})^2 + 2\bar{a}\sum_{p\in\mathcal{P}}e_p^{t-1}\right)
\end{aligned}$$

$$= \frac{1}{(n-1)}\left(\frac{1}{|\mathcal{P}|}\left(\sum_{p\in\mathcal{P}} e_p^{t-1}\right)^2 - \sum_{p\in\mathcal{P}}(e_p^{t-1})^2\right)$$
$$= \frac{1}{(n-1)\,|\mathcal{P}|}\left(\left(\sum_{p\in\mathcal{P}} e_p^{t-1}\right)^2 - |\mathcal{P}|\sum_{p\in\mathcal{P}}(e_p^{t-1})^2\right)$$

Considering that $m(\sum_{k=1}^{m} a_k^2)-(\sum_{k=1}^{m} a_k)^2 = \sum_{1\leq k<l\leq m}(a_k - a_l)^2$, which is derived from the Lagrange's identity (see Section 2.6.1 of [102])[2] by assuming $b_k = 1$, we get:

$$\theta_t^2 - \theta_{t-1}^2 = -\frac{1}{(n-1)\,|\mathcal{P}|}\sum_{k,l\in\mathcal{P},k<l}(e_k^{t-1} - e_l^{t-1})^2 \tag{5.4}$$

Therefore $\theta_t^2 \leq \theta_{t-1}^2$. □

Lemma 5.5 (Convergence)**.** *The estimate e_i converges to the network wide average $\bar{a}$:*

$$\lim_{t\to\infty} e_i^t = \bar{a}, \forall_{i\in\mathcal{V}}$$

Proof. From Lemma 5.2 the sum of all estimates will remain constant along time. Now, attending to Lemma 5.4, and assuming that all nodes will have an equal opportunity to execute the protocol (all nodes execute the algorithm infinitely many times), let's show that $\lim_{t\to\infty} \theta_t^2 = 0$. By contradiction, suppose that $\lim_{t\to\infty} \theta_t^2 = c_1, c_1 > 0$. This non zero variance can only occur if there are at least two different estimates in the network. Then, there must exist at least two neighbors that also have a different estimate and $|e_i - e_j| \geq c_2$, $c_2 > 0$. But, since all nodes execute the algorithm infinitely often, after one of those neighbors executes the algorithm $e_i = e_j$ (according to Lemma 5.3), and according to Equation 5.4 there is a variance reduction of at least $\frac{1}{2(n-1)}c_2^2$ that would lead the global variance to eventually become lower than any non zero c_1 value that is arbitrated. Since $\lim_{t\to\infty} \theta_t^2 = 0$, then $\lim_{t\to\infty} e_i^t = \bar{a}, \forall_{i\in\mathcal{V}}$. □

This proves the convergence to the correct network wide average at all nodes by executing FU over the considered model. Along time, the estimates produced at all nodes will get closer to the correct result, reducing the global variance at each round. In more detail, at each iteration the variance reduction $\theta_t^2 - \theta_{t-1}^2$ is given by Equation 5.4. It is clear that the variance reduction depends on the differences between the estimates of the participating nodes. It is also obvious that the variation is always negative, or

[2]Lagrange's identity: $(\sum_{k=1}^{m} a_k^2)(\sum_{k=1}^{m} b_k^2) - (\sum_{k=1}^{m} a_k b_k)^2 = \sum_{1\leq k<l\leq m}(a_k b_l - a_l b_k)^2$, holding for real numbers.

equal to zero if there is no differences between the estimates of the participating nodes, i.e. $e_k^{t-1} = e_l^{t-1}, \forall_{k,l \in \mathcal{P}}$. In particular, the bigger the difference between the involved estimates, the bigger will be the variance reduction. This suggests that better results (faster convergence) should be obtained, if we manage to choose at each round the set of participants (node that executes the algorithm and neighbors) that exhibits the largest differences between their estimates, in order to maximize the variance reduction (Equation 5.4).

Before analyzing the correctness of FU in concurrent settings, the impact of message loss will be discussed. For a matter of simplicity, a non concurrent model is still considered. In particular, it will be argued by case analysis that although a message loss might introduce a variance reduction error, it will be immediately nullified by the successful reception of the next flow update message over that link, not compromising the correctness of the algorithm (i.e., convergence to the correct network wide average).

5.2.2.1 Message Loss

Here, the impact of message loss in the correctness of the FU algorithm is analyzed, considering the previous non concurrent execution model. To this end, message exchange can no longer be abstracted. In the particular case of Algorithm 2, messages are required to get the estimates for the node that executes the algorithm and further transmit the resulting flows to its neighbors. Here, it is assumed that two types of messages are exchanged between nodes: one to get the estimates e_j and flows $F_j(i)$ of the neighbors at the beginning of a round, and another to update the flow of each neighbor at the end of the round. For a matter of simplicity, in order to avoid nodes from using outdated estimates during the averaging process, it is considered that each time a node is chosen to execute the algorithm, it is always able to get the updated estimate and corresponding flows of all its neighbors (i.e., successful reception of the first type of message). Although unrealistic, seeming like an unfair strategy to counteract message loss, we will later see that this assumption can be dropped (see Section 5.2.3.1). However, it is assumed that messages with the flow update data might be lost; in other words, only the instruction of line 11 of Algorithm 2 may fail. This means that in the case of a message loss, the target neighbor j will not update its flow with the value computed by i, breaking the flow skew symmetry (Lemma 5.1) and consequently mass conservation (Lemma 5.2).

At first sight and after the previous shallow explanation, message loss seems to affect the correctness of the algorithm, and indeed the variance reduction at each round (Lemma 5.4) is no longer verified. This can be shown by a simple example.

Example 5.1. Consider two nodes a and b with initial input values $v_a = 1$ and $v_b = 3$, belonging to a connected network of n nodes with a global average $\bar{a} = 0$. The only neighbor of a is b, but b is connected to the rest of the network by at least another neighbor. Node a is chosen at round t to execute the algorithm. Let's consider that at round $t-1$ all the flows of both nodes were set to 0, consequently the estimates used in the averaging process are equal to their input values. The resulting average computed by a will be $a_a = 2$, and the flow toward b will be $f_{ab} = -1$. If node b successfully receives the message from a with the new flow, it will set $f_{ba} = 1$, then both will produce the new estimate $e_a = e_b = 2$ and the variance will decrease ($\theta_t^2 - \theta_{t-1}^2 < 0$, i.e. $(2^2 + 2^2) - (3^2 + 1^2) < 0$), as stated by Lemma 5.4. However, if the message is lost, then only node a will produce the new estimate $e_a = 2$ and b will produce its previous one $e_b = 3$. Consequently, in this case the variance will actually increase ($\theta_t^2 - \theta_{t-1}^2 > 0$, i.e. $(3^2 + 2^2) - (3^2 + 1^2) > 0$).

An important thing here is that, although the sum of the whole estimates may change, due to message loss, the amount of change is counterbalanced by the sum of all network flows. As estimates are always a function of the flows (and initial value), at each node the flow values are always consistent with its estimate (line 13 of Algorithm 2). In particular, at each link the variation caused by a message loss is given precisely by the difference between the flow values of the sender and receiver. In fact, taking flows into consideration, the global mass is always conserved as shown by the rewritten mass conservation property (Lemma 5.6). This means that "mass" is never lost, unlike what happens with other averaging approaches [71].

Lemma 5.6 (Rewritten Mass Conservation)**.** *At the end of each round:*

$$M = \sum_{i \in \mathcal{V}} v_i = \sum_{i \in \mathcal{V}} e_i + \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{D}_i} F_i(j)$$

Proof. Let the global mass be $M = \sum_{i \in \mathcal{V}} v_i$, assuming that v_i does not changes along time. Now, consider the sum of all estimates at a given round, i.e. $\sum_{i \in \mathcal{V}} e_i$, substi-

tuting each estimate e_i by the estimation function used to compute it (line 13 of of Algorithm 2), we get:

$$\begin{aligned}
&\sum_{i\in\mathcal{V}} e_i = \sum_{i\in\mathcal{V}}\left(v_i - \sum_{j\in\mathcal{D}_i} F_i(j)\right) \\
\Leftrightarrow &\sum_{i\in\mathcal{V}} e_i = \sum_{i\in\mathcal{V}} v_i - \sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{D}_i} F_i(j) \\
\Leftrightarrow &\sum_{i\in\mathcal{V}} e_i + \sum_{i\in\mathcal{V}}\sum_{j\in\mathcal{D}_i} F_i(j) = \sum_{i\in\mathcal{V}} v_i
\end{aligned}$$

□

As previously described by Example 5.1, the loss of a flow update message may lead to the increase of the global variance, i.e. $\theta_t^2 - \theta_{t-1}^2 > 0$. However, it will be shown that this increase is only temporary, and it is reverted by the successful execution of the algorithm by one of the nodes belonging to the failing link (including the successful reception of the flow update message). In particular, the occurrence of message loss at round t might introduce what is defined as a *variance reduction error* δ^t.

Definition 5.2 (Variance Reduction Error)**.** *It is the difference between the expected variance $\langle\theta_t^2\rangle$ if no message had been lost, and the observed variance with message loss $\langle\theta_t^2\rangle_{\mathcal{L}}$ (i.e., $\mathcal{L}$ nodes fail to receive the flow update message), at a round $t > 0$:*

$$\delta^t = \langle\theta_t^2\rangle - \langle\theta_t^2\rangle_{\mathcal{L}}$$

The value of this error depends on the difference between the new estimate (i.e., average a_i^t) that should be produced and the estimate effectively yield (i.e., previous estimate e_l^{t-1}), for the nodes $\mathcal{L}$ that fail to receive the update message. More precisely, it depends on the distance of those estimates to the correct average $\bar{a}$.

Lemma 5.7 (Value of Variance Reduction Error)**.** *At each round $t > 0$, considering the set $\mathcal{L}$ of neighbors which fail to receive the flow update message from the node i that executes the algorithm and computed the new estimate (i.e., average a_i), the variance reduction error δ^t is given by:*

$$\delta^t = \frac{1}{n-1}\left(|\mathcal{L}|\,(a_i^t - \bar{a})^2 - \sum_{l\in\mathcal{L}}(e_l^{t-1} - \bar{a})^2\right)$$

Proof. Recall that, the variance θ_t^2 at round t is given by $\frac{1}{n-1}\sum_{v\in\mathcal{V}}(e_v^t - \bar{a})^2$, where n denotes the number of nodes, e_v^t the estimate of a node v at round t, and $\bar{a}$ the correct global average (i.e., Equation 5.3). Now, let's separate the set $\mathcal{P}$ of nodes that participate in the algorithm, including the executing node i (i.e., $\mathcal{P} = \mathcal{D}_i \cup \{i\}$), in two subsets: the set $\mathcal{S}$ that contains the nodes that successfully update their state (i.e., node i and neighbors that successfully receive the flow update message), and the set $\mathcal{L}$ which is composed by neighbors that fail to receive the message update for node i. Then, according to the previous set partitioning and Lemma 5.3, the expected variance without faults $\langle\theta_t^2\rangle$ is given by $\frac{1}{n-1}(\sum_{p\in\mathcal{P}}(a_i^t-\bar{a})^2+\sum_{j\in\mathcal{V}\setminus\mathcal{P}}(e_j^t-\bar{a})^2)$, and the variance with message loss $\langle\theta_t^2\rangle_{\mathcal{L}}$ by $\frac{1}{n-1}(\sum_{s\in\mathcal{S}}(a_i^t-\bar{a})^2+\sum_{l\in\mathcal{L}}(e_l^{t-1}-\bar{a})^2+\sum_{j\in\mathcal{V}\setminus\mathcal{P}}(e_j^t-\bar{a})^2)$. The result follows by taking the differences between these two terms. □

δ^t can have a positive or negative value, depending if $\langle\theta_t^2\rangle_{\mathcal{L}}$ is lesser or greater than $\langle\theta_t^2\rangle$, which means that the variance can actually improve due to message loss and not always increases (if $\langle\theta_t^2\rangle_{\mathcal{L}} < \langle\theta_t^2\rangle$). In particular, from Lemma 5.7, at each node that fails to receive the update message, if the distance between the previous estimate and the global average (i.e., $(e_l^{t-1} - \bar{a})^2$) is greater than the distance between the expected estimate without loss and $\bar{a}$ (i.e., $(a_i^t - \bar{a})^2$), then the introduced error is negative and increases the global variance, otherwise the inverse is observed.

In front of that, it is now clear that the variance may no longer decrease monotonically in each round, when message loss is taken into account. However, this does not mean that the algorithm will no longer converge. In particular, in order to allow the convergence of the algorithm, it seems that the variance reduction errors introduced by message loss must be nullified along time, and its "correction" can not setback the progress (convergence) of the nodes estimate towards the global average. Beside this, some additional fairness properties must also be met, namely to allow all nodes to eventually participate in the protocol and successfully receive some flow update messages.

Now let's examine in more detail what are the circumstances in which the variance reduction error is nullified, and what scenarios will still allow the convergence of the algorithm towards the global average. In particular, consider that a node i executes the algorithm at time t, and the set $\mathcal{P}$ of participants includes all its neighbors (i.e., $\mathcal{P} = \mathcal{D}_i \cup \{i\}$). At the end of the round, only a subset $\mathcal{S}$ of the participants, excluding i, correctly updates its state with the resulting flow, and a subset $\mathcal{L}$ fails to receive the flow update message (i.e., $i \notin \mathcal{S} \cup \mathcal{L}$, $\mathcal{D}_i = \mathcal{S} \cup \mathcal{L}$, and $\mathcal{L} = \mathcal{D}_i \setminus \mathcal{S}$). Moreover, it

is assumed that up to round t no message was lost. For illustrating purposes and to aid understanding the effect of message loss, the performed analysis is materialized by Example 5.2 which concretely defines the studied message loss scenario.

Example 5.2. Consider five nodes a, b, c, d and e with respective initial input values $v_a = 1$, $v_b = 3$, $v_c = 1$, $v_d = 3$ and $v_e = 4$. These nodes are part of a connected network of size n and with a global average $\bar{a} = 0$. Node a, c and d have three neighbors, respectively $\mathcal{D}_a = \{b, c, d\}$, $\mathcal{D}_c = \{a, d, e\}$, $\mathcal{D}_d = \{a, c, e\}$. Node e has two neighbors $\mathcal{D}_e = \{c, d\}$, and b is the node that connects to the rest of the network, having also a in its neighborhood (i.e., $a \in \mathcal{D}_b$). At round $t = 1$, node a is chosen to execute the algorithm, and at that time the flows of all considered nodes are equal to 0, consequently their estimates are equal to their input values. In this case, the average computed by a is $a_a = 2$, which will be set as the new estimate at all neighbors that successfully receive the respective flow update message, and those that fail to receive the message will keep their state unchanged. This scenario is depicted by Figure 5.1, where it is visible the effect of message loss (from node a to nodes b and d, at the bottom-right) when compared with a situation without loss (at the bottom-left). In order to facilitate the visualization, the estimates are represented by a bar graph with the area of the bars proportional to the estimate value of each node, and flows are depicted by another plot with directed arrows of length equivalent to each flow value (to help visualize the amount added/removed to the initial value, yielding the current node estimates). Figure 5.1 clearly illustrates the introduction of a variance reduction error due to the loss of the flow update message from a to b and d, breaking the skew symmetry between node a and its neighbors b and d.

Now, let's consider that in the next round $t + 1$ node i or one of its neighbors (i.e., $s \in \mathcal{S} \vee l \in \mathcal{L}$) are chosen to execute the algorithm, and analyze what happens:

- **[i is chosen]** $\Rightarrow$ the new average a_i^{t+1} is computed using the following estimate values: for all nodes belonging to $\mathcal{S}$ the estimate will be equal to the average resulting from the previous round (i.e., $\forall s \in \mathcal{S}, e_s^t = a_i^t$, according to Lemma 5.3); for all nodes in $\mathcal{L}$ their previous estimate e_i^{t-1} is used (i.e., $\forall l \in \mathcal{L}, e_l^t = e_l^{t-1}$), since they did not receive the previous flow update message; node i will use the flows associated to each received estimate to yield its estimate (i.e., e_i' from line

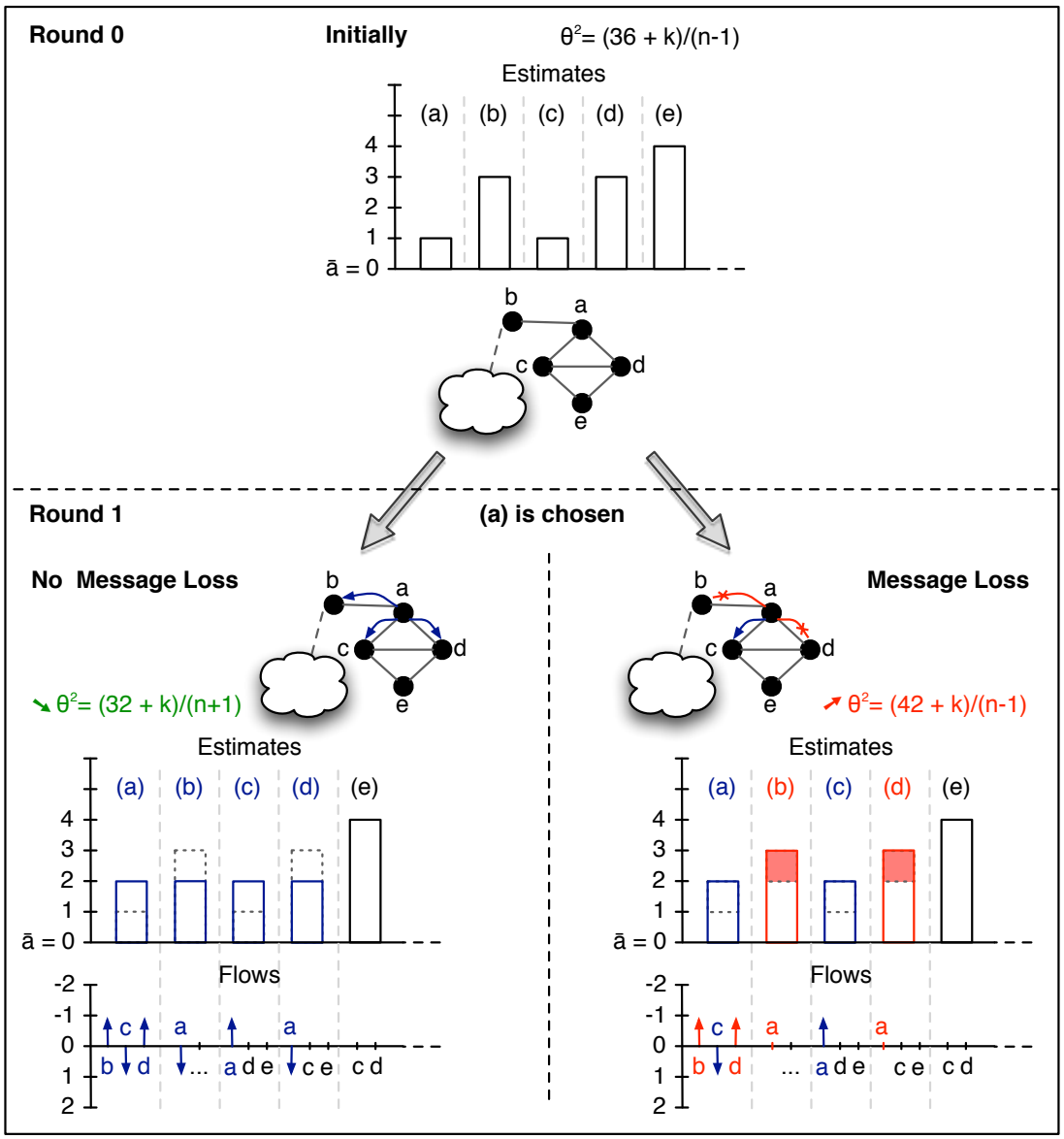


Figure 5.1: Example (non concurrent model) depicting the effect of message loss.

7 of Algorithm 2), which will likely be different from the one resulting from the previous round (because outdated flow values will be used from elements of $\mathcal{L}$, although unchanged values will be used for members of $\mathcal{S}$). More precisely, considering the different flow values used by each set $\mathcal{S}$ and $\mathcal{L}$, the estimate $e_i^{t+1'}$ for node i is given by:

$$e_i^{t+1'} = v_i + \sum_{s \in \mathcal{S}} F_s(i)^t + \sum_{l \in \mathcal{L}} F_l(i)^t$$

The flows of nodes in $\mathcal{S}$ are correctly updated, then $F_s(i)^t = -F_i(s)^t$. On the other hand, the flows of nodes in $\mathcal{L}$ fail to be updated, and $F_l(i)^t = F_l(i)^{t-1}$. Since, it assumed that until round t no message has been lost, then at round $t-1$, $F_l(i)^{t-1} = -F_i(l)^{t-1}$. From these observations it follows that:

$$e_i^{t+1'} = v_i - \sum_{s \in \mathcal{S}} F_i(s)^t - \sum_{l \in \mathcal{L}} F_i(l)^{t-1}$$

As the flows of nodes in S result from the computation performed at line 10 of Algorithm 2, then:

$$\begin{aligned} e_i^{t+1'} &= v_i - \sum_{s \in \mathcal{S}} \Big(F_i(s)^{t-1} + (a_i^t - e_s^{t-1}) \Big) - \sum_{l \in \mathcal{L}} F_i(l)^{t-1} \\ &= v_i - \sum_{s \in \mathcal{S}} F_i(s)^{t-1} - \sum_{l \in \mathcal{L}} F_i(l)^{t-1} - |\mathcal{S}|\, a_i^t + \sum_{s \in \mathcal{S}} e_s^{t-1} \end{aligned}$$

According to the estimation function, line 13 of Algorithm 2:

$$e_i^{t+1'} = e_i^{t-1} - |\mathcal{S}|\, a_i^t + \sum_{s \in \mathcal{S}} e_s^{t-1}$$

Since no message is lost before round t, then $e_i^{t'} = e_i^{t-1}$. Then, from the average computation, line 8 of Algorithm 2, we can derive the expression $e_i^{t-1} + \sum_{s \in \mathcal{S}} e_s^{t-1} = |\mathcal{P}|\, a_i^t - \sum_{l \in \mathcal{L}} e_l^{t-1}$. Substituting the sum of the estimates of i and all $s \in \mathcal{S}$ in the above equation, we get:

$$\begin{aligned} e_i^{t+1'} &= |\mathcal{P}|\, a_i^t - \sum_{l \in \mathcal{L}} e_l^{t-1} - |\mathcal{S}|\, a_i^t \\ &= a_i^t + |\mathcal{L}|\, a_i^t + |\mathcal{S}|\, a_i^t - \sum_{l \in \mathcal{L}} e_l^{t-1} - |\mathcal{S}|\, a_i^t \\ &= a_i^t + |\mathcal{L}|\, a_i^t - \sum_{l \in \mathcal{L}} e_l^{t-1}; \end{aligned} \tag{5.5}$$

Recapitulating, the new average a_i^{t+1} will be computed using a_i^t as the estimate for all nodes $s \in \mathcal{S}$, the previous estimate e_l^{t-1} for all nodes $l \in \mathcal{L}$, and finally the value given by Equation 5.5 as the estimate $e_i^{t+1'}$ of i. Therefore, the new

average will be:

$$\begin{aligned} a_i^{t+1} &= \frac{\sum_{s\in\mathcal{S}} e_s^t + e_i^{t+1'} + \sum_{l\in\mathcal{L}} e_l^t}{|P|} \\ &= \frac{|\mathcal{S}|\, a_i^t + a_i^t + |\mathcal{L}|\, a_i^t - \sum_{l\in\mathcal{L}} e_l^{t-1} + \sum_{l\in\mathcal{L}} e_l^t}{|\mathcal{P}|} \\ &= \frac{|\mathcal{P}|\, a_i^t}{|\mathcal{P}|} \\ &= a_i^t \end{aligned}$$

In this case, the new average will be equal to the one computed in the previous round (i.e., $a_i^{t+1} = a_i^t$), and i will have another opportunity to transmit the same update messages to all its neighbors. Moreover, if all nodes that previously failed to receive the message succeed this time, then the global state will be equal to the one found if no message have been lost. In this last situation, it is easy to observe that the global variance varies (i.e., may decrease, $\theta_{t+1}^2 \leq \theta_{t-1}^2$) as if no message has been lost, despite the previous perturbation in the variance (i.e., possible increase, $\theta_t^2 \geq \theta_{t-1}^2$). In this specific case, message loss has only delayed the convergence. Example 5.3 illustrated a concrete example of this situation.

Example 5.3. This example starts from the situation of message loss considered by Example 5.2 (i.e., a is chosen to execute the algorithm and the flow update messages to b and d are lost), and a is again chosen to execute the algorithm in the next round. Figure 5.2 closely depicts the performed computation, separating the estimate and flow values used as input of the averaging process (left) from the resulting values at the end of the round (right). In this situation, a gets outdated estimates and corresponding flows from b and d, which will be responsible for the modification of its estimate, and the values associated to c are unchanged (values from c are equal to the ones held at a). The flows between (a, b), and (a, d) will became skew symmetric, reducing the estimate used by a (to compute the average) by an amount (-2) which is equal to the sum of the estimation error resulting from the previous message loss at node b and d $(1 + 1)$. Therefore, the computed average will be equal to the previous one, since the participants and sum of their estimates are the same. The variance reduction error introduced by the previous message loss at b and d is nullified, if now those nodes successfully receive the flow up-

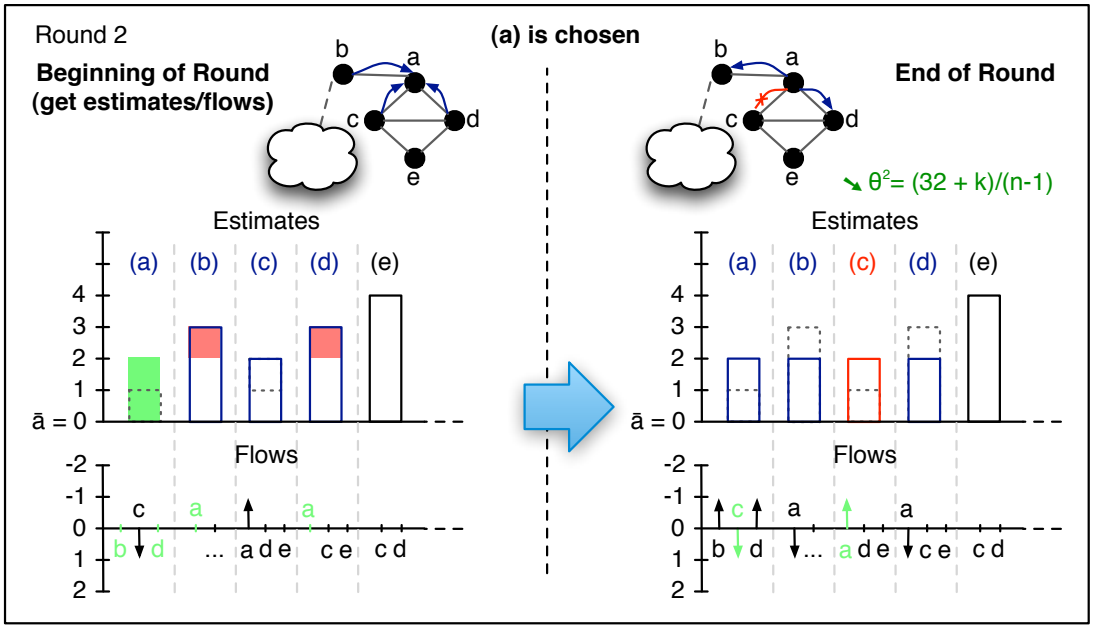


Figure 5.2: Example (non concurrent model) where the same node is chosen to execute the algorithm after message loss.

date message. Note that, no flow update message will be required toward c (message can be lost), as the newly computed average is equal to its previous estimate (successfully updated in the previous round).

- **[$l \in \mathcal{L}$ is chosen]** $\Rightarrow$ as it is assumed that the node that executes the algorithm is able to get the estimate and correspondent flow from all its neighbors, then node l will get the lost data from the previously chosen node i. It will be as if l has successfully received the flow update message from i, and the variance reduction error corresponding to this link (i, l) will be nullified. A concrete situation is showed by Example 5.4. Note that other members of $\mathcal{L}$ (neighbors of l) may participate, but the variance reduction error introduced by them for losing the message from i will be maintained. This is verified because the execution of the algorithm by l does not change the flow values between those nodes and i, as flows are associated to links (i.e., at each node separated flow variables are used for each neighbor).

Example 5.4. This example starts from the situation of message loss considered by Example 5.2 (i.e., a is chosen to execute the algorithm and the flow

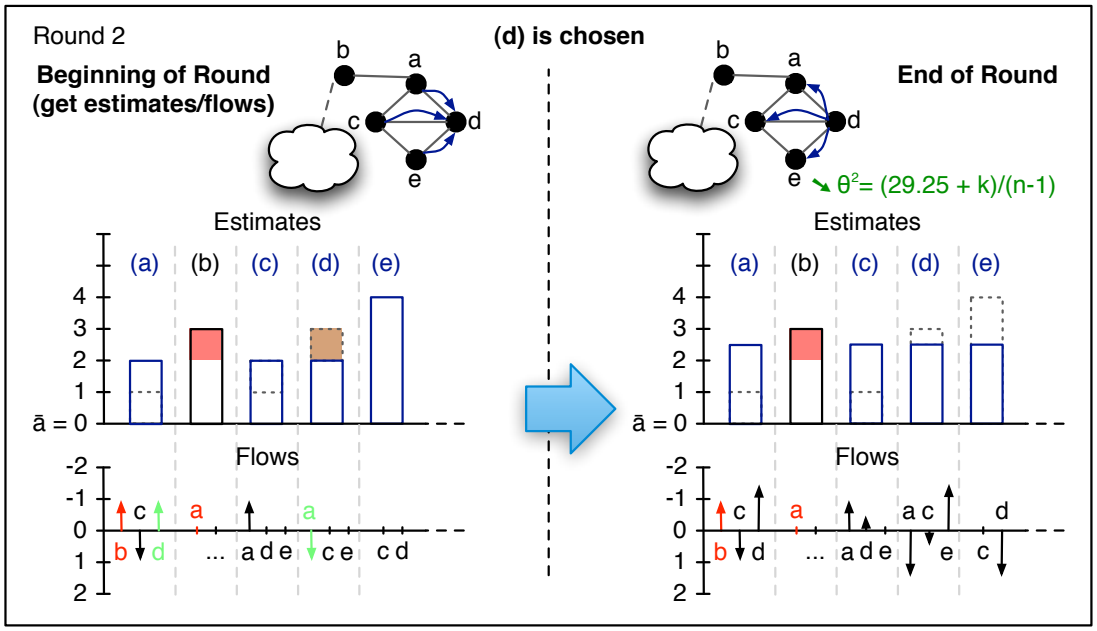


Figure 5.3: Example (non concurrent model) where a node that fails to receive a message from the previous round is chosen to execute the algorithm.

update messages to b and d are lost), and $d \in \mathcal{L}$ is chosen to execute the algorithm in the next round. Figure 5.3 depicts the execution of the algorithm, separating the estimate and flow values used as input of the averaging process from the resulting values (at the end of the round). In this case, d gets the estimate and flows that it was supposed to receive in the previous round, and the execution will proceed as if the message from a has not been lost. Notice that only the variance reduction error originated by the message lost in the link between a and d can be nullified (if the flow update message is successfully delivered), because only the skew symmetry of this link is restored at the beginning of the round.

- **[$s \in \mathcal{S}$ is chosen]** $\Rightarrow$ it is clear that, if none of the neighbors of s has previously lost the flow update message (independently from having participated in the previous round), then they will contribute to the variance reduction according to Lemma 5.4, assuming that all will successfully receive the respective flow update message. However, the amount of variance reduction error added by previous message loss will not be changed and still be part of the global variance. The same happens if some node l belonging to $\mathcal{L}$ participates in the algorithm,

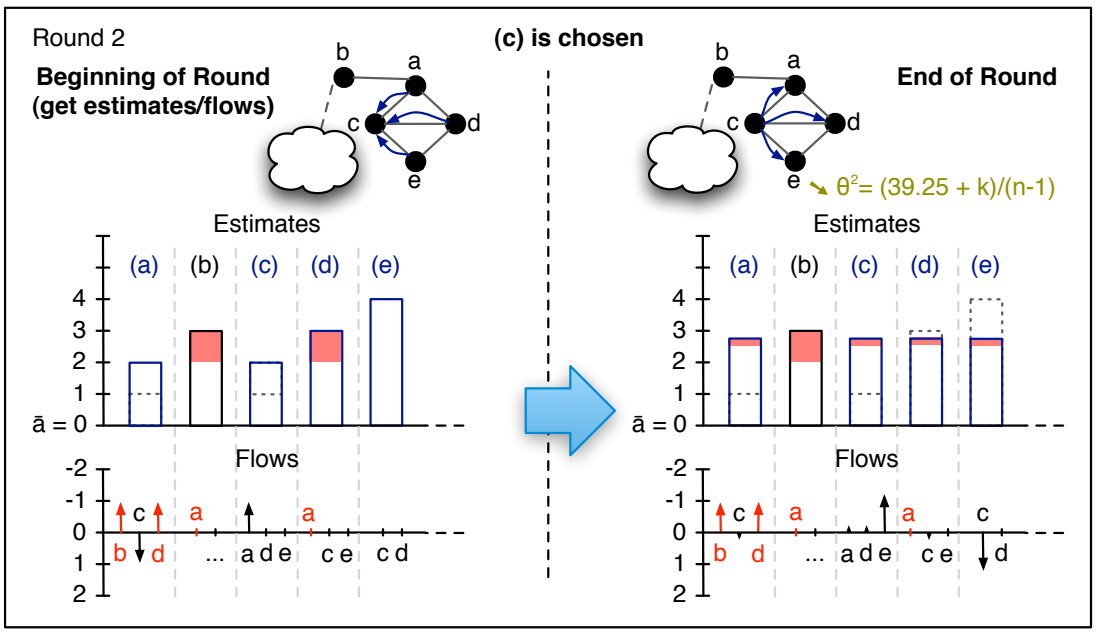


Figure 5.4: Example (non concurrent model) where a node that successfully received a message from the previous round is chosen to execute the algorithm.

because s will only use and update the flow associated to the link between them and not between i and l. Example 5.5 illustrates a specific situation. Note that, an identical behavior is observed if another node j that did not participate in the previous round of the algorithm (i.e., $j \in \mathcal{V} \backslash \mathcal{P}$) is now chosen, see Figure 5.5.

Example 5.5. This example starts from the situation of message loss considered by Example 5.2 (i.e., a is chosen to execute the algorithm and the flow update messages to b and d are lost), choosing $c \in \mathcal{S}$ to execute the algorithm in the next round. Figure 5.4 depicts the execution of the algorithm, separating the estimate and flow values used at the beginning of the round (left) from the resulting values at the end of the round (right). In this example, c gets the estimate and flows from its neighbors, and the estimate c uses for the averaging process is not changed (corresponding to the one at the end of the previous rounds). A new average is computed and if all flow update messages are successfully delivered, then all participants will produce the new estimate and the global variance will decrease (compared to its previous value). However, the same variance reduction error is still present, since no action is taken between nodes on links where a message was previously lost (i.e., flow values

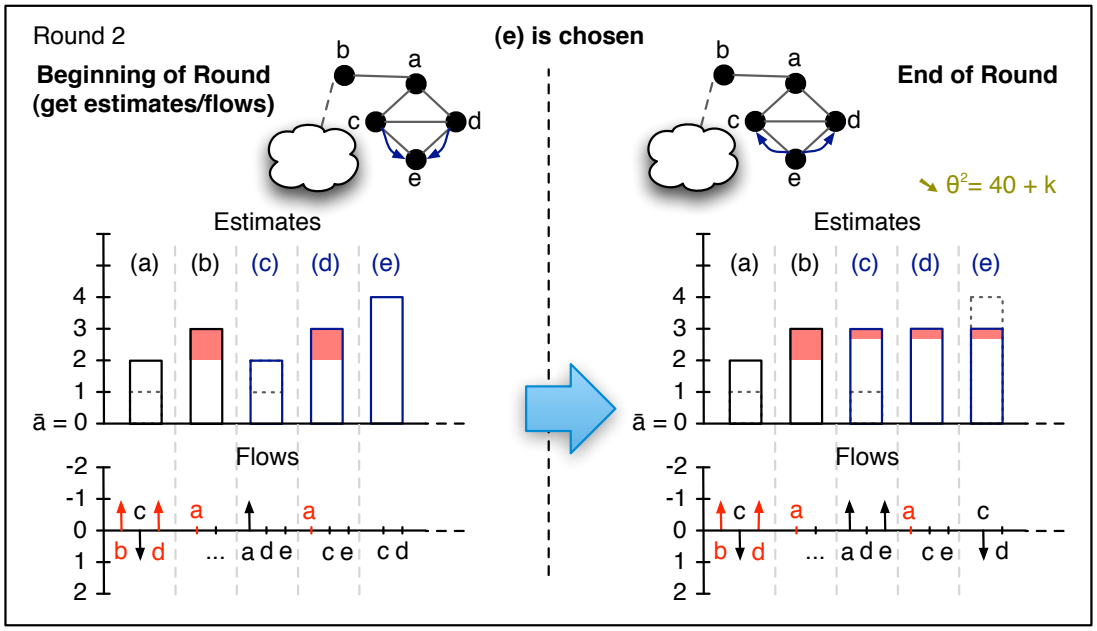


Figure 5.5: Example (non concurrent model) where a node that did not participate in the previous round is chosen to execute the algorithm.

associated to (a, b) and (a, d) are unchanged). Note that, the same behavior is observed if node e is chosen to execute the algorithm, which did not participate in the previous round, see Figure 5.5. The variance reduction error will only be reduced if a node belonging to a link where a previous message loss has occurred is chosen to execute the algorithm, and the flow update message is successfully delivered on this link (restoring the flow symmetry). This last situation can be observed in Figure 5.6, where node d is chosen to execute the algorithm in a third round, after choosing node e in the previous one.

In summary, a message loss introduces a (temporary) variance reduction error that in a sense is associated to the link where it occurred. At each faulty link, the value of the reduction error depends only on the difference between the previous estimate at the node that failed to receive the update message and the estimate (group average) that it was supposed to produce (more precisely, the difference between the distance of those estimates to the correct average, i.e. $(a_i^t - \bar{a})^2 - (e_l^{t-1} - \bar{a})^2$). At a time t, the sum of the reduction errors at all links will yield the variance reduction error δ^t given by Lemma 5.7. From the performed analysis, we observe that the error introduced at each faulty link is nullified by a successful delivery of a flow update message, resulting from the round execution of the algorithm by any of the two nodes at the extremities

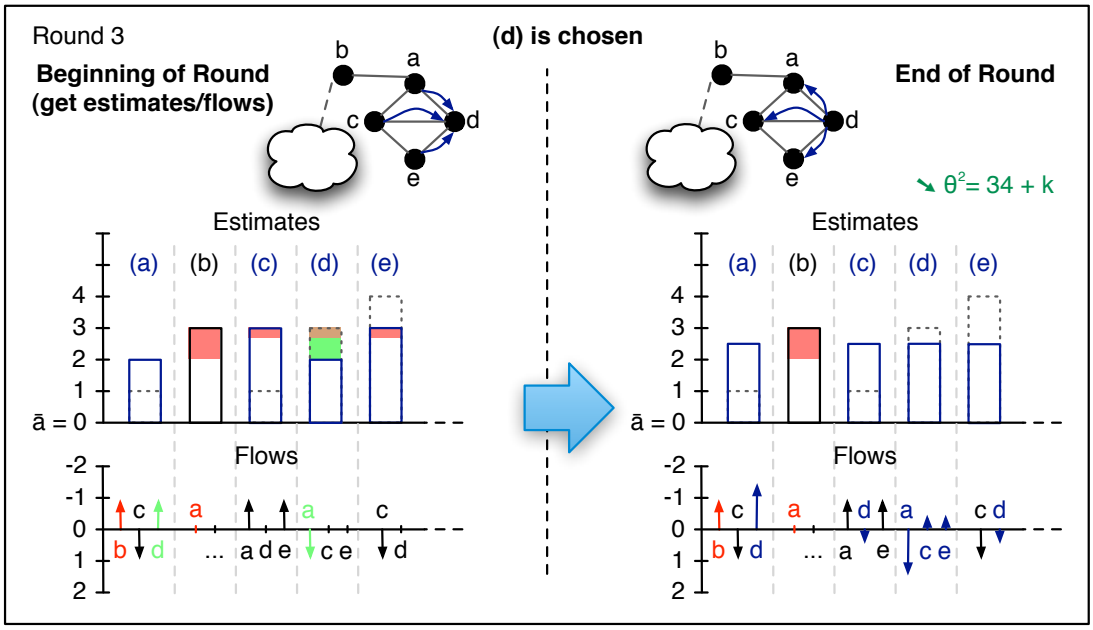


Figure 5.6: Example (non concurrent model) where the node chosen to execute the algorithm belongs to a link where a message was previously lost.

of the concerned link. Furthermore, it was also observed that, even without nullifying any component of the variance reduction error, other round executions which involved the successful delivery of new flow updates (with an estimate different from the previous one) will contribute to the progress of the algorithm (i.e., decrease the variance). Therefore, if it is given an opportunity to all nodes to execute the algorithm and successfully transmit flow update messages, keeping all nodes in the network reachable (i.e., the same node is not always chosen to execute the algorithm, and the messages in the same link are not always lost), then the estimate at all nodes will converge to the network wide average.

In conclusion, message loss does no affect the correctness of the algorithm, only delaying the convergence of the algorithm to the global average, as long as some fairness is ensured (e.g., no communication channel fails forever, infinite number of message are successfully transmitted after message loss, and all nodes are chosen to execute the algorithm an infinitely many times). As the estimates at all nodes converge to the same value along time, the difference between them and the distance to the global average decreases, and so does the variance reduction error due to message loss, which tends to zero.

5.2.3 Concurrent Model (with non overlapping groups)

In the previous section, the correctness of *Flow Updating* (i.e., convergence at all nodes to the global average) has been proven on a (simple) non concurrent model. However, the previous model is not practical and is slow, as only one node exclusively executes the algorithm at each round. Therefore, in this section the correctness of the algorithm is analyzed considering the concurrent execution of FU. In particular, the convergence at all nodes to the correct average is proved, based on the previous reasoning, as long as a mutual exclusion condition is met (to avoid concurrent flow updates on the same edge).

It is quite obvious that it will be advantageous to admit the concurrent execution of the algorithm by several distinct nodes at the same time. Then, consider that each individual execution involves the participation of a set of nodes $\mathcal{P}_i$, generally composed by the node i that effectively executes the algorithm and some of its neighbors (i.e., $\forall_{i,j \in \mathcal{P}_i}\ j \in \mathcal{D}_i$), forming what will be referred as an *averaging group*. At this point, for a matter of simplicity, let's assume that each node can only be involved in one iteration at the same time (i.e., $\forall_{i \in \mathcal{P}_j}\ i \notin \mathcal{P}_k$ for $j \neq k$), in other words the averaging groups do not overlap (i.e., $\mathcal{P}_j \cap \mathcal{P}_k = \{\}$, for $j \neq k$). In fact, in these settings the concurrent execution of FU by g non overlapping groups at the same round t, each one executing a single instance led by different nodes, will yield the same result as the consecutive execution of the algorithm by each one of those groups along g successive rounds. Therefore, guaranteeing at each round that the averaging groups do not overlap is a sufficient condition to maintain the correctness of FU (convergence). In this case, the previous correctness analysis holds, and the variance reduction observed at a single round in the non concurrent model (Equation 5.4) can be significantly improved. Now, corresponding to the sum of the variance reductions resulting from the execution of the algorithm by each concurrent group, where g is the number of (non overlapping) groups, and $\mathcal{P}_k$ is the set of participants belonging to each group k:

$$\theta_t^2 - \theta_{t-1}^2 = \sum_{k=1}^{g} \left(-\frac{1}{(n-1)\,|\mathcal{P}_k|} \sum_{i,j \in \mathcal{P}_k, i<j} (e_i^{t-1} - e_j^{t-1})^2 \right) \tag{5.6}$$

Equation 5.6 suggests that the variance reduction can be increased by choosing the set of participants that have estimates with more distant values, as already pointed in the previous section. Consequently, it appears advantageous for any averaging algorithm to consider this property, when deciding which nodes should be included in each

group. A version of FU that takes advantage of this observation is proposed in Section 5.3.1, which is called *Flow Updating with Preferential Grouping* (FUPG).

In order to analyze the correctness of FU on a concurrent execution model, a slightly modified version of the previous algorithm is considered, which is detailed by Algorithm 3. This algorithm also abstracts the communication between nodes, and the averaging step is represented by the procedure $averaging(\mathcal{P}_i)$ (line 12-16) which corresponds to the core of Algorithm 2. Here, the main difference is that instead of an implicit fixed group of participants, always composed by all the neighbors of the node that executes the algorithm, now the set of participants is taken as a parameter, since it can change. It is also assumed that each node can only participate in a single instance of the algorithm at each round, so that the averaging groups do not overlap. This means that different neighbors of a specific node can participate in distinct averaging groups at the same time, although each are constrained to participate in a singe group at each round. In practice this can be easily implemented, for instance: each node can simply choose a leader (i.e., one of its neighbor or itself) to send its estimate at each round, and participate in the iteration leaded by the chosen node; nodes that receive at least one estimate execute the algorithm (i.e., averaging step), being set as leader of the source node. This practicality is regarded by Algorithm 3 (line 6-11).

The correctness of this algorithm can be proved using basically the same reasoning made for Algorithm 2 (see Section 5.2.2), but considering that the group of participants in each averaging step can vary (instead of always including all neighbors in each iteration). More specifically, it can be observed that, according to the previous assumptions, the concurrent execution of several instances of FU at distinct averaging groups in the same round ends up to be equivalent to the sequential execution of each individual group along consecutive rounds (i.e., yields the same result). Considering this equivalence between concurrent executions and sequential non concurrent executions, the correctness of FU holds, but some technicalities must be taken into consideration. In particular, a mutual exclusion condition between leaders must be met and some lemmas must be rewritten to match the considered model.

First, it easy to show that the leader assignment scheme used in Algorithm 3 guaranties the assumption that averaging groups do not overlap.

Lemma 5.8 (Non-Overlapping Averaging Groups)**.** *At any round $t > 0$, for any two different sets of nodes, $\mathcal{P}_j^t$ and $\mathcal{P}_k^t$ with $j \neq k$, that participate in the algorithm at the*

1 **inputs:**
2 v_i, value to average
3 $\mathcal{D}_i$, set of known neighbors
4 **state variables:**
5 flows: initially, $F_i = \{j \leftarrow 0 \mid j \in \mathcal{D}_i\}$
6 leader: initially, $l_i = i$
7 **algorithm:**
8 $\mathcal{P}_i = \{j \mid j \in \mathcal{D}_i \cup i \wedge l_j = i\}$
9 **if** $\mathcal{P}_i \neq \{\}$ **then**
10 $averaging(\mathcal{P}_i)$
11 $l_i = decideLeader(\mathcal{D}_i \cup i)$
12 **averaging($\mathcal{P}_i$):**
13 $a_i = \dfrac{\sum_{j \in \mathcal{P}_i} e_j}{|\mathcal{P}_i|}$
14 **forall the** $j \in \mathcal{P}_i \backslash \{i\}$ **do**
15 $F_i(j) = F_i(j) + (a_i - e_j)$
16 $F_j(i) = -F_i(j)$
17 **estimation function:**
18 $e_i = v_i - \sum_{j \in \mathcal{D}_i} F_i(j)$

Algorithm 3: Flow Updating algorithm, considering concurrent executions at distinct nodes, and abstracting message exchanges.

same round:

$$\mathcal{P}_j^t \cap \mathcal{P}_k^t = \{\}.$$

Proof. Suppose there is an element i that belongs to both $\mathcal{P}_j^t$ and $\mathcal{P}_k^t$. From the definition of averaging groups (line 8 of Algorithm 3), for $i \in \mathcal{P}_j^t$ we have $l_i = j$, and for $i \in \mathcal{P}_k^t$ we have $l_i = k$. But this contradicts $j \neq k$. □

Now, let's examine the skew symmetry property of the algorithm, which is important to guaranty the conservation of the global mass of the system, and it is a basilar property used in the proof of other lemmas. In this particular case, the skew symmetry requires an additional condition to hold, namely that no two nodes choose each other as leader.

Property 5.1 (Mutual Exclusion)**.**

$$\forall_{j \in \mathcal{D}_i}, l_i = j \Rightarrow l_j \neq i$$

Property 5.1 can be ensured in practice by simple local decisions at each node, e.g. setting itself as leader when a conflict is detected (i.e., an estimate is received from the chosen leader), although restraining the participation of the nodes in the desired group (see Section 5.3.1).

Lemma 5.9 (Rewritten Skew Symmetry)**.** *At any round, for all $i \in \mathcal{V}$:*

$$\textit{Property 5.1 implies } F_i(j) = -F_j(i), \forall_{j \in \mathcal{D}_i}$$

Proof. Each node i only writes its own flow variables $F_i(j)$, and those of other neighbors $F_j(i)$, for $j \in \mathcal{P}_i$ (line 15-16 of Algorithm 3). The only case when $F_i(j) \neq -F_j(i)$ is if two neighbors i and j are both leaders, with $j \in \mathcal{P}_i$ and $i \in \mathcal{P}_j$. Therefore, if Property 5.1 holds, then $F_i(j) = -F_j(i)$. □

Unlike in the previous non concurrent model where the node that executes the averaging process always participates in it (including its estimate), in this case the node that leads the averaging process may not participate in it (if it chooses another leader). Therefore, in order to prove the flow adjustment lemma, in particular considering that a node does not participate in the group it leads, the notion of *flow conservation* was introduced. Basically, if the node is not included, flow conservation establish that the sum of the flows of all neighbors participating in an averaging process is maintained, not affecting the estimate of the executing node.

Lemma 5.10 (Flow Conservation)**.** *At any round $t > 0$, considering the set of participants $\mathcal{P}_i^t$ in the averaging process executed by a node i at round t, and that $i \notin \mathcal{P}_i^t$, we have:*

$$\sum_{j \in \mathcal{P}_i^t \setminus \{i\}} F_i(j)^t = \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} F_i(j)^{t-1}.$$

Proof. At any round $t > 0$, each node i calculates the average of the set of nodes $\mathcal{P}_i^t$ which elected it, and computes the flow values for them to produce the new average as estimate (line 12-16 of Algorithm 3). The flow values at the each leader are set according to line 15 of Algorithm 3, i.e. $F_i(j)^t = F_i(j)^{t-1} + (a_i^t - e_j^{t-1})$, for all participants except the leader itself (i.e., $\forall_{\mathcal{P}_i^t \setminus \{i\}}$, line 14). Recall that, at a given round, each

node only chooses one leader, and two neighbors can not choose each other (Property 5.1), guarantying that no concurrent flow updates occur on the same link. Therefore, considering the sum of all flows updated at node i at round t, we get:

$$\begin{aligned}\sum_{j\in\mathcal{P}_i^t\setminus\{i\}} F_i(j)^t &= \sum_{j\in\mathcal{P}_i^t\setminus\{i\}} \left(F_i(j)^{t-1} + (a_i^t - e_j^{t-1})\right) \\ &= \sum_{j\in\mathcal{P}_i^t\setminus\{i\}} F_i(j)^{t-1} + \left|\mathcal{P}_i^t\setminus\{i\}\right| a_i^t - \sum_{j\in\mathcal{P}_i^t\setminus\{i\}} e_j^{t-1}\end{aligned}$$

From the average computation (line 13), and assuming that i does not participate (not elected itself), i.e. $i \notin \mathcal{P}_i^t$, we derive $|\mathcal{P}_i^t\setminus\{i\}|\, a_i^t = \sum_{j\in\mathcal{P}_i^t\setminus\{i\}} e_j^{t-1}$. Then, substituting in the above expression the result follows. □

Now, it is possible to show that the flow adjustment property is verified independently from the inclusion of the leader in the averaging group.

Lemma 5.11 (Rewritten Flow Adjustment)**.** *At the end of any round $t > 0$, considering the estimate e_j^t of all participants $\mathcal{P}_i^t$ in the averaging process executed by a node i, independently if $i \in \mathcal{P}_i^t$ or $i \notin \mathcal{P}_i^t$:*

$$\forall_{j\in\mathcal{P}_i^t}, e_j^t = a_i^t$$

Proof. At each round t, only leaders can update flows, and the estimate of each node can only change as a result those updates.

First, let's determine the estimate of a node j that chooses a node i different from itself as leader (i.e., $j \in \mathcal{P}_i^t$). Notice that, j can also be chosen as leader by a subset of its neighbors $\mathcal{P}_j^t\setminus\{j\}$, except from i according to the condition set by Property 5.1. Therefore, the estimate of j is obtained from the flow update performed by its leader i, the flows updated by itself for its averaging group $\mathcal{P}_j^t\setminus\{j,i\}$, and the unchanged flows of its remaining neighbors $\mathcal{D}_j\setminus\mathcal{P}_j^t \cup \{i\}$. Consequently, separating those flows in the estimation function (line 18 of Algorithm 3), we get:

$$e_j^t = v_j - F_j(i)^t - \sum_{k\in\mathcal{P}_j^t\setminus\{j,i\}} F_j(k)^t - \sum_{l\in\mathcal{D}_j\setminus\mathcal{P}_j^t\cup\{i\}} F_j(l)^t$$

Concerning the flow update performed by i, from Lemma 5.9 $-F_j(i)^t = F_i(j)^t$ and as $F_i(j)^t = F_i(j)^{t-1} + (a_i^t - e_j^{t-1})$ (line 15), replacing $F_j(i)^t$ we obtain:

$$e_j^t = v_j + F_i(j)^{t-1} + a_i^t - e_j^{t-1} - \sum_{k \in \mathcal{P}_j^t \setminus \{j,i\}} F_j(k)^t - \sum_{l \in \mathcal{D}_j \setminus \mathcal{P}_j^t \cup \{i\}} F_j(l)^t$$

Given that: according to Lemma 5.9 $F_i(j)^{t-1} = -F_j(i)^{t-1}$; as j is not included in the averaging process it executes, from Lemma 5.10 the sum of the flow of its participants is conserved, i.e. $\sum_{k \in \mathcal{P}_j^t \setminus \{j,i\}} F_j(k)^t = \sum_{k \in \mathcal{P}_j^t \setminus \{j,i\}} F_j(k)^{t-1}$; since the flows of the neighbors that have not elected or been elected by j are unchanged, $\forall_{l \in \mathcal{D}_j \setminus \mathcal{P}_j^t \cup \{i\}} F_j(l)^t = F_j(l)^{t-1}$. Then, replacing the flows attending to these observations, we get:

$$\begin{aligned} e_j^t &= v_j - F_j(i)^{t-1} + a_i^t - e_j^{t-1} - \sum_{k \in \mathcal{P}_j^t \setminus \{j,i\}} F_j(k)^{t-1} - \sum_{l \in \mathcal{D}_j \setminus \mathcal{P}_j^t \cup \{i\}} F_j(l)^{t-1} \\ &= v_j - F_j(i)^{t-1} - \sum_{k \in \mathcal{P}_j^t \setminus \{j,i\}} F_j(k)^{t-1} - \sum_{l \in \mathcal{D}_j \setminus \mathcal{P}_j^t \cup \{i\}} F_j(l)^{t-1} + a_i^t - e_j^{t-1} \end{aligned}$$

Taking into consideration the initial flow separation, the estimate of j at round $t-1$ is $e_j^{t-1} = v_j - F_j(i)^{t-1} - \sum_{k \in \mathcal{P}_j^t \setminus \{j,i\}} F_j(k)^{t-1} - \sum_{l \in \mathcal{D}_j \setminus \mathcal{P}_j^t \cup \{i\}} F_j(l)^{t-1}$, then:

$$\begin{aligned} e_j^t &= e_j^{t-1} + a_i^t - e_j^{t-1} \\ &= a_i^t \end{aligned}$$

Now, let's consider the estimate of a node i that chose itself as leader. In this case, following the previous reasoning, i.e. separating the flows to the participating nodes from those of the remaining neighbors, the estimate of the leader i is given by:

$$e_i^t = v_i - \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} F_i(j)^t - \sum_{k \in \mathcal{D}_i \setminus \mathcal{P}_i^t} F_i(k)^t$$

Since the flow of all participating neighbors $\mathcal{P}_j^t$ are computed by i from $F_i(j)^t = F_i(j)^{t-1} + (a_i^t - e_j^{t-1})$ (line 15), then:

$$e_i^t = v_i - \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} \left(F_i(j)^{t-1} + a_i^t - e_j^{t-1} \right) - \sum_{k \in \mathcal{D}_i \setminus \mathcal{P}_i^t} F_i(k)^t$$

$$= v_i - \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} F_i(j)^{t-1} - \left|\mathcal{P}_i^t - 1\right| a_i^t + \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} e_j^{t-1} - \sum_{k \in \mathcal{D}_i \setminus \mathcal{P}_i^t} F_i(k)^t$$

Attending that the flows of the neighbors that did not elected i are unchanged, i.e. $\forall_{k \in \mathcal{D}_i \setminus \mathcal{P}_i^t} F_i(k)^t = F_i(k)^{t-1}$, we get:

$$\begin{aligned} e_i^t &= v_i - \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} F_i(j)^{t-1} - \sum_{k \in \mathcal{D}_i \setminus \mathcal{P}_i^t} F_i(k)^{t-1} - \left|\mathcal{P}_i^t - 1\right| a_i^t + \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} e_j^{t-1} \\ &= e_i^{t-1} - \left|\mathcal{P}_i - 1\right| a_i^t + \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} e_j^{t-1} \end{aligned}$$

Finally, from the average computation (line 13), including the estimate of i, we derive $\left|\mathcal{P}_i^t\right| a_i^t = \sum_{j \in \mathcal{P}_i^t \setminus \{i\}} e_j^{t-1} + e_i^{t-1}$. Then, substituting the sum of estimates, we obtain:

$$\begin{aligned} e_i^t &= \left|\mathcal{P}_i^t\right| a_i^t - \left|\mathcal{P}_i^t - 1\right| a_i^t \\ &= a_i^t \end{aligned}$$

□

Taking into consideration the previously revised properties, it is straightforward to demonstrate that the variance reduction of the non concurrent model (Lemma 5.4) holds for each averaging group of the concurrent model. More specifically, the total variance reduction at each round is given by Equation 5.6. Consequently, the convergence of the estimates at all nodes to the global average also holds in this setting (Lemma 5.5), and the proof can be performed by the same reasoning used in the non concurrent model. This proves the correctness of FU in the considered concurrent model without faults. Next, the impact of message loss considering concurrent executions of FU is discussed.

Note that, if we consider that in each round all nodes choose themselves and all its neighbors as leaders, the averaging groups will overlap, and we will get a fully concurrent version of the algorithm similar to the one initially detailed at the beginning of this chapter (i.e., Algorithm 1). However, as already explained at the start of this section, we were unable to reach a correctness proof in a fully concurrent setting, and leave this problem for future work. Nevertheless, all empirical evaluations have confirmed the correctness of the main FU algorithm, exhibiting the best performance, especially in terms of robustness (see Chapter 7).

5.2.3.1 Message Loss

Previously, in Section 5.2.2.1, the effect of message loss on the execution of FU in non concurrent settings was explained in detail. However a strong assumption was made on the acquisition of estimates from neighbors, assuming that nodes that execute the algorithm are always able to get the estimates of all its neighbors. In others words, messages to get estimates were never lost, only messages to update flows. Now, this assumption is relaxed, and this kind of message can also be lost, which is quite realistic. In particular, it is assumed that estimates and flows are both carried in a single type of message, sent by each node at the end of the algorithm (immediately after line 11 of Algorithm 3). Moreover, besides estimates and flows, the information of the chosen leader is also sent in the same message. Taking this into consideration, let's analyze the impact of message loss on the correctness of FU in the defined concurrent execution model.

Notice that the information of chosen leaders is used at the beginning of the algorithm (line 8) for each node to decide if it will execute the averaging step (when elected by at least one of its neighbors), i.e. to set the participants of its averaging group. Therefore, if some node fails to receive the information that it has been chosen as leader, due to message loss, it might lose the opportunity to start an averaging step or execute it with less participants (including only neighbors from which it has successfully receive the message in the previous round). As estimates are sent in the same message as the leader information, the estimates of all participants will always be available at the leader. Consequently, at this level message loss only induces a loss of opportunity for some neighbors to participate in the leader averaging process. Clearly, this loss of opportunity only reduces the total number of participants (i.e size and number of averaging groups) at each round, decreasing the variance reduction (Equation 5.6), without impacting the correctness (only delaying convergence).

While on one hand, the lost of the estimate (and chosen leader) will not introduce any error, the same cannot be said in terms of flows. In fact, among the nodes that participate in a given round, i.e. those that successfully delivered their messages at the end of the previous round, the effect of message loss in terms of flow updates (line 16) is similar to the one previously described in Section 5.2.2.1. Namely, at the flow level, message loss might introduce a variance reduction error (see Definition 5.2 and Lemma 5.7). However, this error is immediately nullified at each link upon the successful reception of a further message, without compromising the convergence of

the algorithm to the correct global average. Therefore, as in the non concurrent model, in this settings message loss does not impact the correctness of FU, as long as some fairness is met.

5.3 Variations and Improvements

In this section, some variations and improvements to the original *Flow Updating* algorithm are described. Namely, a version that implements the previous concurrent algorithm is proposed, which is called Flow Updating with Preferential Grouping (FUPG). Additionally, some practical extensions to the initial algorithm are also introduced, in order to address termination and the execution of FU on asynchronous settings.

5.3.1 Flow Updating with Preferential Grouping

Attending to the observations of Section 5.2.3, a new version of FU was designed: *Flow Updating with Preferential Grouping* (FUPG). The algorithm can be simply seen as an implementation of the previously described Algorithm 3 in a synchronous model, where nodes decide to participate in averaging groups according to some heuristic that tries to maximize the variance reduction of each averaging process. To this end, the concept of *reduction potential* is introduced, which is a measure of the possible variance reduction achieved by including a specific set of nodes in the averaging step. This value is computed at each round by every node, and used to set the participants of each averaging group in the next round.

FUPG is detailed by Algorithm 4. This algorithm creates averaging groups that intend to maximize the resulting variance reduction, using two main functions: *computeRP* (line 35), and *decideLeader* (line 34). The first function simply computes a value representing the expected variance reduction (i.e., *reduction potential*, line 8) of a node, that will be used by the second function (in the next round) to decide in which group the node will participate, more precisely to choose a leader. More importantly, the algorithm must guaranty mutual exclusion on the access to flows over the same link (between chosen leaders), i.e., two neighbors cannot choose each other as leader in the same round (Property 5.1) in order to ensure the skew symmetry of flows of the participants of an averaging group (see Lemma 5.9).

This property is ensured at each node i simply by checking if the chosen leader l_i is included in the set of participants P (i.e., nodes that have chosen i as leader, line

inputs:
  v_i, value to average
  $\mathcal{D}_i$, set of known neighbors
state variables:
  flows: initially, $F_i = \{j \mapsto 0 \mid j \in \mathcal{D}_i\}$
  participants: initially, $P_i = \{\}$
  leader in the next round: initially, $l_i = i$
  reduction potential: initially, $r_i = initRP()$
  estimates: initially, $E_i = \{j \mapsto 0 \mid j \in \mathcal{D}_i\}$
message-generation function:
  $\text{msg}_i(F_i, P_i, l_i, r_i, E_i, j) = (i, f, \text{est}(v_i, F_i), l_i, r_i)$
  with $f = \begin{cases} F_i(j) & \text{if } j \in P_i \\ \perp & \text{otherwise} \end{cases}$
state-transition function:
  $\text{trans}_i(F_i, P_i, l_i, r_i, E_i, M_i) = (F_i', P_i', l_i', r_i', E_i')$ **with**
  $F = \{j \mapsto -f \mid (j, f, \_, \_, \_) \in M_i \wedge f \neq \perp\} \cup$
    $\{j \mapsto f \mid (j, \perp, \_, \_, \_) \in M_i \wedge (j, f) \in F_i\} \cup$
    $\{j \mapsto f \mid (j, \_, \_, \_, \_) \notin M_i \wedge (j, f) \in F_i\}$
  $E = \{j \mapsto e \mid (j, \_, e, \_, \_) \in M_i \wedge (j, \_) \notin P_i\} \cup$
    $\{j \mapsto e \mid (j, \_, \_, \_, \_) \in M_i \wedge (j, \_) \in P_i \wedge (j, e) \in E_i\} \cup$
    $\{j \mapsto e \mid (j, \_, \_, \_, \_) \notin M_i \wedge (j, e) \in E_i \wedge j \neq i\} \cup \{i \mapsto \text{est}(v_i, F)\}$
  $P = \{j \mid (j, \_, \_, l, \_) \in M_i \wedge l = i\}$
  $l = l_i$
  if $l \neq i \wedge l \in P$ **then** `/* ensure Property 5.1 */`
    if $id(i) > id(l)$ **then**
      $P = P \backslash \{l\}$
    else
      $P = P \cup \{i\}$
      $l = i$
  if $P \neq \{\}$ **then**
    $a = \sum_{j \in P} E(j) / |P|$
    foreach $j \in P \backslash \{i\}$ **do**
      $F(j) = F(j) + (a - E(j))$
      $E(j) = a$
  if $l \neq i$ **then** $l_i' = i$ **else** $l_i' = decideLeader(M_i, r_i, P)$
  $r_i' = computeRP(E, M_i, l)$
  $F_i' = F; P_i' = P; E_i' = E$
estimation function:
  $\text{est}(v, F) = v - \sum\{f \mid (\_, f) \in F\}$

Algorithm 4: Flow Updating with Preferential Grouping algorithm.

21) and removing the leader from the set P in that case (line 23-28). More precisely, in this case upon detection of a leader conflict, the leader is removed if it has a lower ID that the node itself (lines 24-25) or set to the node itself otherwise (lines 26-28), allowing both of them to participate in the same averaging process, performed by one of the leader. In alternative, both could remove the other from the set of participants and set the leader to themselves, but then none will participate in the averaging step of the other at that time. Note that no global UID is required, only local identifiers to distinguish neighbors (that can be totally ordered).

Nevertheless, in this case, satisfying Property 5.1 is not enough to ensure the correctness of the algorithm, in particular to ensure that the estimates used as input in the next averaging group correspond to the ones resulting from the previous group average (after the flow update). This is due to the fact that, unlike in the previous model where each averaging group is created and the resulting flows updated in the same round, in this case the group averaging process may span two rounds: one round, for the leader to receive the estimates from participants, compute the new average and update its flows, and another round, for the leader to send the resulting flow updates to the participants). In this case, except the leader that locally updates its flows in the same round it performs the group averaging, each participant will only produce the estimate resulting from the group average upon the reception of the flow update from the leader in the next round. Therefore, participants must wait for the flow update from their leader, before choosing and sending their estimates to a new one. In order to ensure this consistency between estimates, after choosing a leader different from itself, a node always chooses itself as leader (line 34), so that it receives the flow update from the previous leader and can start using the updated estimate.

At each node i, the essence of this algorithm consist of the computation of the average of the estimates of the nodes (neighbors and self) that chose i as leader, and corresponding flow computation (lines 21-32). The new computed flows are sent in the next round only to the corresponding set of participants P_i (line 12), i.e., $\perp$ is sent otherwise. At the beginning of each round, the flows F are updated with the skew symmetric value received from the previous leader (lines 15-17). In addition to the flows F_i, set of participants P_i, leader l_i, and reduction potential r_i, the most recent estimates of all neighbors E_i is also keep in the state of each node. This is used for two main purposes. First and more important, it holds the updated estimates of the participants in the current averaging group leaded by i. Second, it is used by the heuristics

defined to choose a new leader, more specifically to compute the reduction potential. Next, some of the heuristics defined to create averaging groups are described.

5.3.1.1 Formation of Averaging Groups

Here, two heuristics that can be used for the creation of averaging groups at each round are described. The proposed heuristics take advantage of the previous theoretical analysis of the variance reduction, trying to maximize its value (see Equation 5.6) at each round in a decentralized way. More precisely, these heuristics define the way leaders are chosen, i.e., the implementation of the functions *decideLeader* and *computeRP* of Algorithm 4 (lines 34-35).

The first heuristic is based on the computation of the expected reduction potential at each node (derived from equation 5.4), as if it was chosen as leader by all its neighbors. The node with the higher value will be the one that provides the higher variance reduction if chosen by all its neighbors. The key functions of this heuristic are detailed by Algorithm 5. In this case, the reduction potential corresponds directly to a proportion of the expected variance reduction, which is computed by the function *computeRP* (line 7-8). The function *decideLeader* uses the reduction potentials (line 4) of the node and its neighbors (received from messages) to choose the one with the higher value as leader (line 5). If the maximum value is shared by several nodes, one of them is randomly chosen as leader (line 6).

An alternative heuristic is proposed in Algorithm 6, which is also based on the computation of the expected variance reduction considering the possible participation of all neighbors. The main difference is that it tries to "guess" the average resulting from the leader and anticipate its use to compute the expected variance reduction. In this case, at each node the reduction potential is composed by a pair of values: the expected average (line 12) and variance reduction (line 13). In particular, at each node the expected average of the leader (if different from self) is used as input of the computation of the reduction potential, instead of using the estimate of the leader and the node itself (line 11). The core of the leader decision function is similar to the previous one, i.e., the node with the higher reduction potential is chosen. Nevertheless, some changes are introduced with the intent to improve the performance of the algorithm. Namely, if a node i has chosen itself as leader and has been chosen by at least one of its neighbors j in the previous round, then it will not consider itself as a candidate to become leader in the current round, i.e., its potential reduction data will be removed

initRP():
  return 0
decideLeader$(M, rp, \_)$**:**
  $R = \{j \mapsto r \mid (j, \_, \_, \_, r) \in M\} \cup \{(i, rp)\}$
  $\mathcal{C} = \{j \mid (j, m) \in R \wedge m = \max(\{r \mid (\_, r) \in R\})\}$
  return $random(\mathcal{C})$
computeRP$(E, \_, \_)$**:**
  return $\frac{1}{|E|} \sum_{\substack{(j,\_),(k,\_)\in E \\ j<k}} (E(j) - E(k))^2$

Algorithm 5: Functions used to decide the leader and compute the reduction potential, relying on the expected variance reduction by including all neighbors.

initRP():
  return $(0, 0)$
decideLeader(M, rp, P)**:**
  $R = \{j \mapsto r \mid (j, \_, \_, \_, r) \in M\} \cup \{(i, rp)\}$
  if $|P| > 1$ **then** `/* i ∈ P because l_i = i */`
    $R = R \backslash \{(i, \_)\}$
  $C = \{j \mid (j, (m, \_)) \in R \wedge m = \max(\{r \mid (\_, (r, \_)) \in R\})\}$
  return $random(\mathcal{C})$
computeRP(E, M, l)**:**
  $R = \{j \mapsto r \mid (j, \_, \_, \_, r) \in M\} \cup \{(i, rp)\}$
  `/* snd() takes the second element from a pair */`
  $E(j)' = \begin{cases} snd(R(l)) & \text{if } (j = i \vee j = l) \wedge l \neq i \\ E(j) & \text{otherwise} \end{cases}$
  $a = (\sum\{e \mid (\_, e) \in E'\}) / |E'|$
  $c = \frac{1}{|E'|} \sum_{\substack{(j,\_),(k,\_)\in E' \\ j<k}} (E(j)' - E(k)')^2$
  return (c, a)

Algorithm 6: Functions used to decide the leader and compute the reduction potential, relying on the leader average expectation and variance reduction including all neighbors.

from the list of candidates (lines 5-6). This intends to provide an opportunity for node i to participate at its turn in the averaging process of one of its neighbors j (that will set itself as leader).

The performance of the algorithm using these different heuristics was evaluated, and the results can be found in Section 7.4. Other heuristics could be defined to set the leader of each node, e.g., random assignment of a leader, but from the performed experiments better results were always obtained when using heuristics that attempt to maximize the variance reduction.

5.3.2 Termination/Quiescence

Termination detection is an important feature on the design of most distributed algorithms, preventing them from running forever and spending unnecessary resources. This is the case of Flow Updating, which will run forever if no stoping criteria is defined, infinitely converging toward the correct result. Therefore, in this section a simple strategy is proposed, in order to avoid FU from sending messages forever and waste unnecessary resources upon reaching a defined level of accuracy.

FU can be applied in two main operation modes: *on-demand* mode, one-time execution in response of an explicit aggregation query; *monitoring* mode, continuous execution providing an updated estimate at all nodes (e.g., piggybacked to other protocols). In either of these modes, a stopping condition is required after reaching a given accuracy: in the on-demand mode to terminate the execution of the algorithm and report the result to the aggregation query; in the monitoring mode to enter in a quiescence state, avoiding the transmission of unnecessary messages, only reacting to dynamic changes (i.e., nodes arriving/leaving and change of the input values).

In order to implement termination/quiescence, a straightforward strategy was defined, relying only on the local data available at each node. The termination detection scheme is simply based on the computation of the difference between two consecutive estimates, and verification if it has reach a predefined threshold. In more detail, each node i at the end of each averaging process computes the relative difference ϵ_i^t between the current estimate e_i^t and its previous one e_i^{t-1}, given by:

$$\epsilon_i^t = \frac{\mid e_i^t - e_i^{t-1} \mid}{e_i^t}$$

If the relative difference ϵ_i^t between estimates is smaller than a predefined threshold ξ,

i.e. $\epsilon_i^t < \xi$, then FU stops sending messages to its neighbors, entering a quiescent state, but continues to process all received messages. However, upon the reception of further data from neighbors, it might happen that ϵ_i^t increases above the given threshold ξ, i.e. $\epsilon_i^t \geq \xi$, and therefore node i starts sending messages again, leaving its quiescent state. Since all nodes do not reach a result with a common accuracy at the same time, due to the decentralized nature of the algorithm, an additional parameter ci was considered to account the number of consecutive *iterations* in which ϵ_i^t is smaller than the threshold ξ. Therefore, the execution of the algorithm at each node only effectively terminates or enters a quiescent state, after verifying that $\epsilon_i^t < \xi$ during a predefined number ci of consecutive iterations of the algorithm.

A detailed evaluation of the application of this termination detection strategy in FU is provided in Section 7.6. From the obtained results, it is possible to verify that it suffices to wait a few rounds (e.g., $ci = 1$, $ci = 2$) to reach a good tradeoff between message load and accuracy for the predefined threshold ξ. The use of an additional threshold ζ to leave the quiescent state (with $\zeta > \xi$) was also examined, but no visible advantages were obtained from its use; therefore, it is not considered.

5.3.3 Asynchrony

Up to now, it has been assumed that Flow Updating is executed in synchronous settings, but synchronism is not required for the algorithm to work. In fact, FU was designed to operate on realistic settings, namely dynamic and asynchronous networks with message loss. Nevertheless, some simple modifications to the original algorithm are required in order to operate in asynchronous settings, which are described in this section. Note that synchrony can be implemented over asynchronous fault-free networks (see Chapter 16 of [90]), using a *synchronizer*.

In asynchronous settings, each transmitted message incurs an arbitrary delay and no guaranty can be made on the time of its reception. Moreover, considering message loss, no guaranty can even be made about its delivery. As message do not arrive at a constant pace (like in round based models), the main issue is to determine when to execute the averaging step of FU. After the reception of each message from a neighbor? or wait for the reception of the messages from all neighbors? Due to possible message loss, and in order to avoid nodes from waiting forever for a lost message, a timeout must be set.

In fact, in order for FU to operate in asynchronous settings, a simple timeout strat-

egy is used, waiting for the reception of a message from all neighbors during a predefined time τ to execute the algorithm. More specifically, each node stores all messages received from different neighbors in a buffer, one per neighbor, as an older message is always replaced by the most recent one (assuming FIFO communication channels, otherwise sequential message identifiers can be added), until one of two conditions is met: a message from all neighbors is received, or the timeout τ is reached. Then, an iteration of FU (i.e., averaging step) is executed using the buffered data, a message is sent to all neighbors with the resulting data (i.e., flows and estimate), the message buffer is cleaned and the timeout τ is reset.

An evaluation of the described strategy is available in Section 7.7. The results obtained show that the performance of FU is affected by the defined timeout value, depending on the distribution of the network message latencies. In particular, best results are obtained when the timeout is set to more conservative values (but not too conservative), i.e., with a value greater than the average message transmission time and preferably close to the percentiles 95-99% of the distribution of the message latencies.

Chapter 6

Estimating Complex Aggregates

This chapter introduces a novel distributed algorithm based on Flow Updating to compute "complex" aggregation functions, more precisely to estimate the Cumulative Distribution Function (CDF) of some system wide attribute. The base FU approach previously described is restricted to the computation of scalar aggregation functions (e.g., AVERAGE, COUNT, SUM). However, in some situations the use of a scalar value to represent some global properties might be insufficient or even inadequate to direct the execution of some distributed application, e.g., hiding meaningful outlier values from a decision taking process. In many of those situations, the knowledge of the statistical distribution of the concerned attribute can provide more useful and relevant information, even if less accurate.

For example, consider some load balancing application that aims to distribute equitably the global load of a system. In this case, the knowledge of the total or average load does not provide enough information to assess the distribution of the system load, i.e. determine if some processing nodes are overloaded or idle. Even the computation of the maximum and minimum is insufficient, although it allows the detection of a difference in the global load distribution, as it does not provide information about the number of processes at each load level. In this situation, an estimation of the statistical load distribution is required to provide the desired information and reveal outlier values. Other examples can be found in the context of monitoring applications. For instance, in WSN estimating the distribution of the monitored attribute can be very useful to distinguish isolated sensor anomalies from the occurrence of a relevant event characterized by a certain amount of abnormal values.

More specifically, the proposed algorithm allows the decentralized estimation of

the Cumulative Distribution Function (CDF) of a target attribute. A CDF can be characterized by a set of interpolation points associated to the frequencies of the input values less or equal than each point. More precisely, considering an input value x_i at each node i, the CDF of x can be approximated by a set of k pairs (s, e), where s is an element in the vector $\vec{s}$ of interpolation points and e is the fraction of values less or equal than s (i.e., $e = |\{x_i \mid x_i \leq s\}| / n$). Considering a single pair of the CDF, it is possible to estimate e through a distributed averaging algorithm. Namely, setting 1 as the input value v_i of each node i that satisfies the condition $x_i \leq s$ and 0 otherwise, then the estimate at all nodes will converge to the fraction of nodes that fulfills the previous condition, as result of the execution of a distributed averaging algorithm such as FU. The main idea of the proposed algorithm is based on this observation, and can be seen as a parallel execution of several instances of FU, one for each pair of the CDF.

For short, this algorithm is referred as FUCDF, as its core is based on the application of FU to estimate a CDF. The computation performed at each node i is detailed by Algorithm 7. As can be observed, in essence the algorithm is identical to FU (see Algorithm 1), but applied to vectors instead of scalar values. Namely, the flows F_i map for each neighbor a vector of flows (one for each interpolation point), $\vec{v_i}$ is now a state vector set according to the input value x_i, and the estimation function yields a vector of estimates.

Additionally, a vector of interpolation points $\vec{s_i}$ is defined to characterize the CDF. At each node, the interpolation vector $\vec{s_i}$ is set with equidistant values between the global minimum and maximum, with a maximum size defined by the input parameter k. Initially, the vector $\vec{s_i}$ is set with a single value (i.e., x_i), and it is recomputed at each iteration according to the known global maximum and minimum, by the function *computeIP* (line 19). Therefore, the MAX and MIN aggregation functions are also computed along the execution of the algorithm: the current max_i and min_i at each node (line 9-10) are sent in all messages (line 12), and upon reception of new values the maximum and minimum value are updated accordingly (line 17-18). This simple scheme allows the determination of the global maximum/minimum at all nodes in d rounds (without faults), where d is the diameter of the network graph. Thus, at the end of d rounds the interpolation vector $\vec{s_i}$ will be the same at all nodes, as all compute it using the same function and input parameters.

All vectors involved in the execution of the algorithm are adjusted according to the current $\vec{s_i}'$ by the function *adjust*, e.g., received flows (line 21-22) and estimates

1 **inputs:**
2 x_i, value to aggregate
3 $\mathcal{D}_i$, set of neighbors given by failure detector
4 k, number of interpolation points
5 **state variables:**
6 flows: initially, $F_i = \{\}$ `/* flows vector for each neighbor */`
7 base frequency vector: initially, $\vec{v_i} = [1]$
8 interpolation points: initially, $\vec{s_i} = [x_i]$
9 maximum value: initially, $max_i = x_i$
10 minimum value: initially, $min_i = x_i$
11 **message-generation function:**
12 $\text{msg}_i(F_i, \vec{v_i}, \vec{s_i}, max_i, min_i, j) = (i, \vec{s_i}, max_i, min_i, \vec{f}, \text{est}(\vec{v_i}, F_i))$;
13 **with** $\vec{f} = \begin{cases} \overrightarrow{F_i(j)} & \text{if } (j, \_) \in F_i \\ \vec{0} & \text{otherwise} \end{cases}$
14 **state-transition function:**
15 $\text{trans}_i(F_i, \vec{v_i}, \vec{s_i}, max_i, min_i, M_i) = (F_i', \vec{v_i}', \vec{s_i}', max_i', min_i')$
16 **with**
17 $max_i' = \max(\{max \mid (\_, \_, max, \_, \_, \_) \in M_i\} \cup max_i)$
18 $min_i' = \min(\{min \mid (\_, \_, \_, min, \_, \_) \in M_i\} \cup min_i)$
19 $\vec{s_i}' = computeIP(max_i', min_i')$
20 $\vec{v_i}' = setV(\vec{s_i}')$
21 $F = \{j \mapsto -adjust(\vec{f}, \vec{s}, \vec{s_i}') \mid j \in \mathcal{D}_i \wedge (j, \vec{s}, \_, \_, \vec{f}, \_) \in M_i\} \cup$
22 $\quad \{j \mapsto adjust(\vec{f}, \vec{s_i}, \vec{s_i}') \mid j \in \mathcal{D}_i \wedge (j, \_, \_, \_, \_, \_) \notin M_i \wedge (j, \vec{f}) \in F_i\}$
23 $E = \{i \mapsto \text{est}(\vec{v_i}', F)\} \cup$
24 $\quad \{j \mapsto adjust(\vec{e}, \vec{s}, \vec{s_i}') \mid j \in \mathcal{D}_i \wedge (j, \vec{s}, \_, \_, \_, \vec{e}) \in M_i\} \cup$
25 $\quad \{j \mapsto adjust(\text{est}(\vec{v_i}, F_i), \vec{s_i}, \vec{s_i}') \mid j \in \mathcal{D}_i \wedge (j, \_, \_, \_, \_, \_) \notin M_i\}$
26 $\vec{a} = (\sum\{\vec{e} \mid (\_, \vec{e}) \in E\}) / |E|$
27 $F_i' = \{j \mapsto \vec{f} + \vec{a} - \overrightarrow{E(j)} \mid (j, \vec{f}) \in F\}$
28 **estimation function:**
29 $\text{est}(\vec{v}, F) = \vec{v} - \sum\{\vec{f} \mid (\_, \vec{f}) \in F\}$

Algorithm 7: Algorithm to estimate CDF with Flow Updating (FUCDF).

(line 24-25). In particular, the vector of initial values $\vec{v_i}$ is reset at each round by the function *setV* (line 20), as the interpolation points might change due to an alteration of the known maximum or minimum. These auxiliary functions are defined in more detail in Algorithm 8. In order to keep the interface with the auxiliary functions simple

```
1  computeIP(max, min):
2      if max = min then
3          return [max]
4      else
5          Δ = (max − min)/k
6          s⃗_(k) = max
7          for j = k − 1 to 1 do
8              s⃗_(j) = s⃗_(j+1) − Δ
9          return s⃗
10 setV(s⃗):
11     for j = 1 to k do
12         if x_i ≤ s⃗_(j) then
13             v⃗_(j) = 1
14         else
15             v⃗_(j) = 0
16     return v⃗
17 adjust(u⃗, r⃗, s⃗):
18     foreach s ∈ s⃗ do
19         c = max({r ∈ r⃗ | r ≤ s})
20         v⃗_(s) = u⃗_(c)
21     return v⃗
```

Algorithm 8: Auxiliary functions used in the FUCDF algorithm.

and compact, it is assumed that they can access the inputs of the main algorithm (i.e., k and x_i). The function *setV* simply sets each element of the vector $\vec{v_i}$ with values 1 or 0, according to the value x_i compared to each (new) interpolation point. The function *adjust* implements a simple heuristic to assign the values associated to previous interpolation points to new interpolation values, based on their proximity. In more detail, the function takes a previous vector $\vec{u}$ and its corresponding interpolation vector $\vec{r}$, and for each element of the (new) interpolation vector $\vec{s}$ it assigns the value in $\vec{u}$ indexed by the closest smaller interpolation point c in $\vec{r}$ to the resulting vector $\vec{v}$.

The self-adapting nature of FU at the core of this algorithm enables it to cope with the dynamic adjustment of all involved vectors. In particular, FUCDF supports dynamic network changes (i.e., nodes arriving/leaving), like FU simply by adding/removing the flows data associated to neighbors. Moreover, similarly to FU, it is also able to seamlessly adapt to changes of the input value x_i, in this case simply by re-

computing the vector $\vec{v_i}$ (line 20). This is sufficient to allow FUCDF to operate in setting where the global maximum and minimum do not change. Nonetheless, additional modifications ought to be taken into consideration if the extreme values change due to dynamism, especially if the maximum decreases and the minimum increases. Note that, the following simple heuristic might be adequate to handle this situation: check if there is no difference between the frequencies of two consecutive interpolation points at an extreme (meaning that there is no value between the two), and remove the extreme interpolation point in this case (recomputing the interpolation vector to hold k elements). However, the dynamic adjustment of the system extreme values (i.e., maximum and minimum) is left for future work.

At each node i, the estimated CDF is obtained by pairing the values from $\vec{s_i}$ with the correspondent frequencies in the vector resulting from the estimation function (i.e., est$(\vec{v_i}, F_i)$). Along time, the estimated frequency associated to each interpolation point converge to the correct value, in accordance to the behavior of FU. In fact, it is expected that FUCDF inherits the main characteristics of FU, in terms of convergence, fault-tolerance and self-adaptation to dynamic changes. Recent results [17] seem to corroborate this expectations, but a thorough evaluation is left for future work.

Part IV

Evaluation

Chapter 7

Evaluation

A custom high level simulator, implemented in Java, was used to evaluate Flow Updating (FU). The simulator uses an even-driven implementation to simulate the execution of algorithms in both synchronous and asynchronous network models. Even in synchronous settings, the simulation level is detailed enough to observe the effect of concurrent message exchanges between nodes (i.e., unlike *PeerSim* [69] which abstracts message exchange in cycle-based simulations). Synchronous simulations follow the synchronous execution model defined in Chapter 2 of [90], with all processes executing *message-generation* and *state-transition* events in lock-step. In asynchronous simulations, the transmission time of each messages varies, according to a predefined message latency distribution, leading each process to execute at different points in time.

This chapter presents the results obtained from the empirical evaluation of FU. Several simulation scenarios are taken into consideration to evaluate FU, in order to: compare its performance against state-of-the-art algorithms; assess its fault-tolerance capabilities; analyze its suitability to operate on dynamic settings; and examine its behavior concerning some practical issues aimed to adjust its execution for realistic environments.

The obtained results show that, in general, FU outperforms classic averaging approaches in terms of convergence speed and message load, especially in networks with low average connection degrees. Moreover, FU exhibits a high resilience to message loss and reveals self-adapting capabilities to dynamic changes, being robust. That is shown in the following sections.

7.1 Simulation Settings

This section details the default simulation setting used to perform the presented evaluation. By default simulations are carried according to the synchronous execution model, although asynchronous operation is also considered in Section 7.7. The same aggregation function is computed by all evaluated algorithms, unless stated otherwise: COUNT (determination of the network size). This aggregation function was chosen as it is the one with the worst performance. Convergence speed depends on the initial data distribution across the network, and COUNT represents an extreme scenario where only one node starts with the value 1 and all others with 0. FU will perform better when computing an AVERAGE of uniformly distributed input values. SUM will have the same performance as COUNT, as it is computed by combining AVERAGE and COUNT.

Three different network topologies were considered for simulation purposes: *random*, *2D/mesh*, and *attach*. The random network consists on a connected graph in which all nodes are randomly linked to each other (according to a predefined average degree d), based on the Erdős–Rényi [45] model. The 2D/mesh defines a connected geographical network in which the communication links are established according to a predefined radio communication range, and nodes are placed uniformly at random, modeling an approximation to the topologies occurring in Wireless Sensor Networks (WSN). Finally, the attach corresponds to a connected network generated using a preferential attachment mechanism, based on the Barabási–Albert [12] model, where some nodes end up with a higher number of links compared to the majority of the nodes (having a small degree). This kind of network is representative of some realistic scenarios, such as the Internet (where routers concentrate the majority of the communication links). By default the networks are assumed to be static, i.e., with no link changes and no nodes arriving or leaving, and without faults (e.g., no message loss or node crash). In some specific simulation scenarios dynamism and communication failures are also taken into account.

The main metric used in most simulation scenarios is the *CV(RMSE)* (Coefficient of Variation of the Root Mean Square Error)[1], which express the global accuracy reached by an algorithm. This metric allows the analysis of the speed and message

[1]Root of the mean squared differences between the estimate e_i at each node i and the correct result $\bar{a}$, divided by the correct result: $\frac{1}{\bar{a}}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(e_i-\bar{a})^2}$

load of the tested algorithms, when combined with the proper criteria, respectively: time (number of rounds) and number of messages sent (by each node). Note that, the message load can be interpreted as an approximation to energy expenditure in WSN, as message transmission is often the dominating factor in terms of energy consumption[2].

The results obtained by evaluating FU in different simulation settings are discussed in the next sections. All results for each evaluated scenario are drawn form 30 trials of the execution of the algorithms under identical settings (i.e., each result corresponds to the average from 30 simulations). Distinct randomly generated networks with the same properties (i.e., topology, size, and average degree) are used in each trial. The detailed simulation settings must be considered as the default for each simulation scenario, although some specific custom settings are usually defined (which are described at the beginning of each respective section).

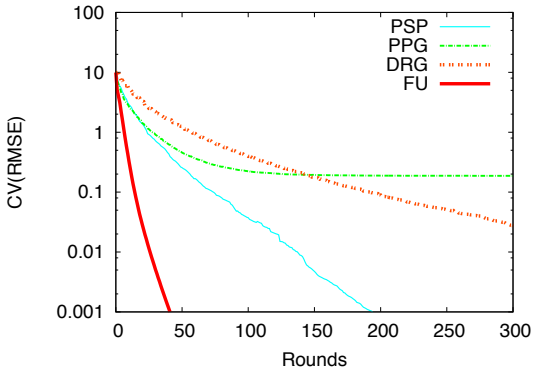
7.2 Comparison Against Other Algorithms

In this section FU is compared to three significant distributed aggregation algorithms from the same class (i.e., averaging): Push-Sum Protocol (PSP) [80], Push-Pull Gossiping (PPG) [68], and Distributed Random Grouping (DRG) [21]. This evaluation is performed under strictly identical simulation settings (same networks and initial distribution of input values), aiming for a fair comparison. In addition, the specific parameters of each algorithm were tuned to obtain the best performance in each simulated scenario (e.g., the probability to become leader in DRG).

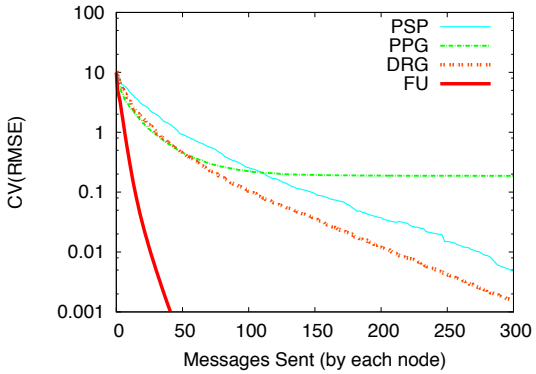
The simulations were performed for the three network topologies (i.e., random, attach, and 2D/mesh), with different sizes (i.e., $n = 100$, $n = 1000$, and $n = 10000$), and distinct average connection degrees (i.e., $d \approx 3$ and $d \approx 10$). The results are depicted by Figures 7.1 and 7.2 for random networks, Figures 7.3 and 7.4 for attach, and Figures 7.5 and 7.6 for 2D/mesh. The first feature observed in all results is that PPG does not converge over time (even without faults). This issue was already reported in Section 4.1.2 and more details can be found in [71].

On random networks with low connection degree (i.e., $d \approx 3$) FU clearly outperforms the other compared algorithms, both in terms of convergence speed and message load. However, a degradation of the performance of FU is observed in networks

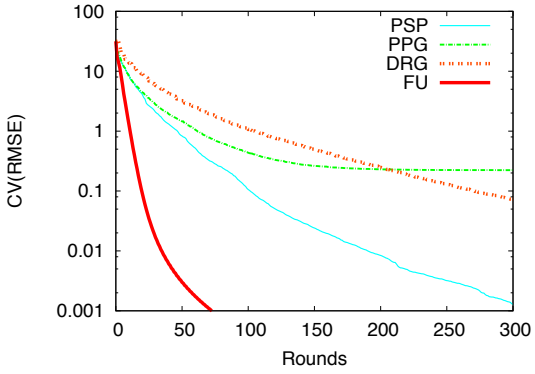
[2]As referred in [7], the energy consumed to transmit a single bit corresponds roughly to the one required to execute thousands of instructions.



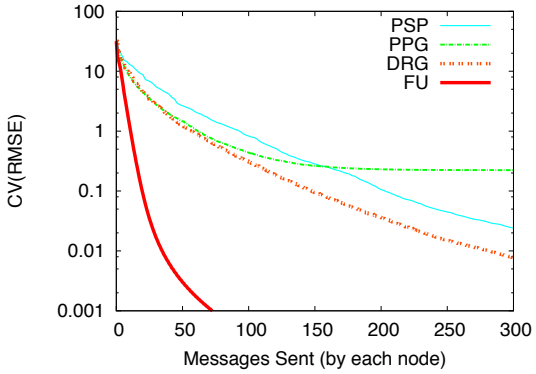
(a) Convergence speed ($n = 100$)



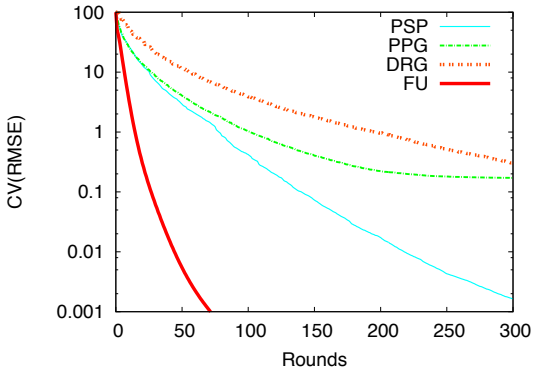
(b) Message load ($n = 100$)



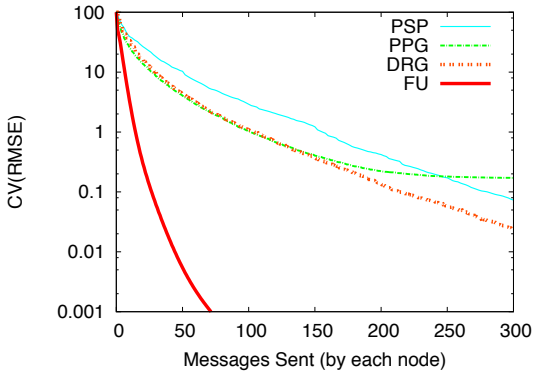
(c) Convergence speed ($n = 1000$)



(d) Message load ($n = 1000$)

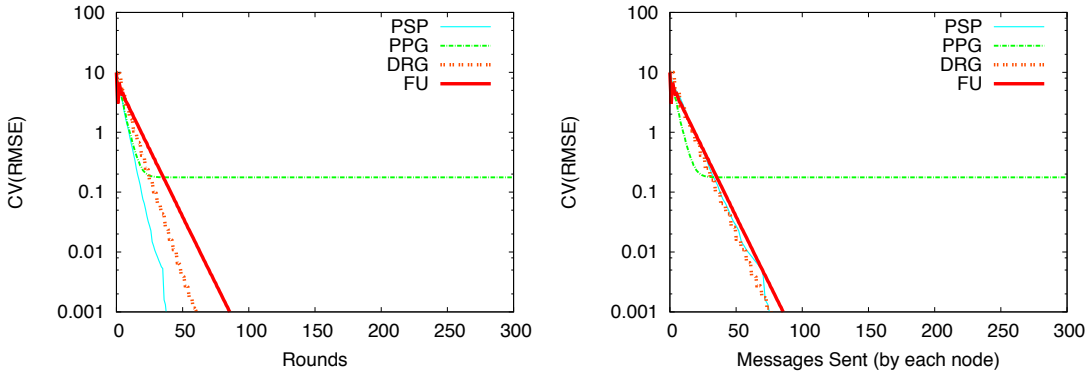


(e) Convergence speed ($n = 10000$)



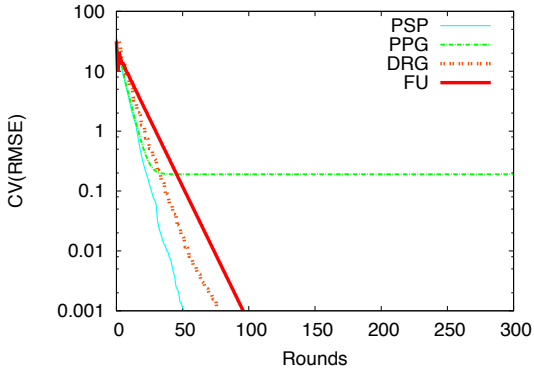
(f) Message load ($n = 10000$)

Figure 7.1: Comparison of *Flow Updating* against other averaging algorithms, on random networks with $d \approx 3$ and different sizes n (i.e., 100, 1000, 10000).

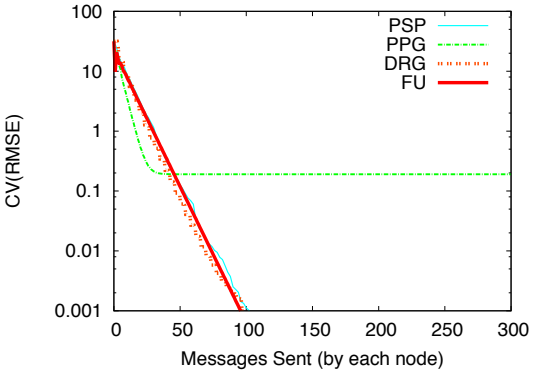


(a) Convergence speed ($n = 100$)

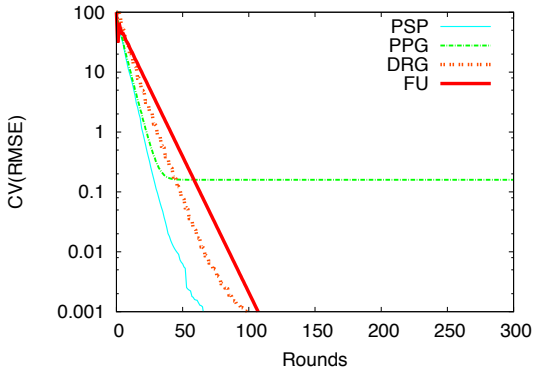
(b) Message load ($n = 100$)



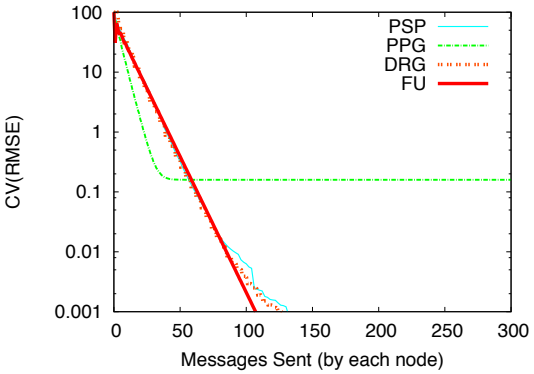
(c) Convergence speed ($n = 1000$)



(d) Message load ($n = 1000$)

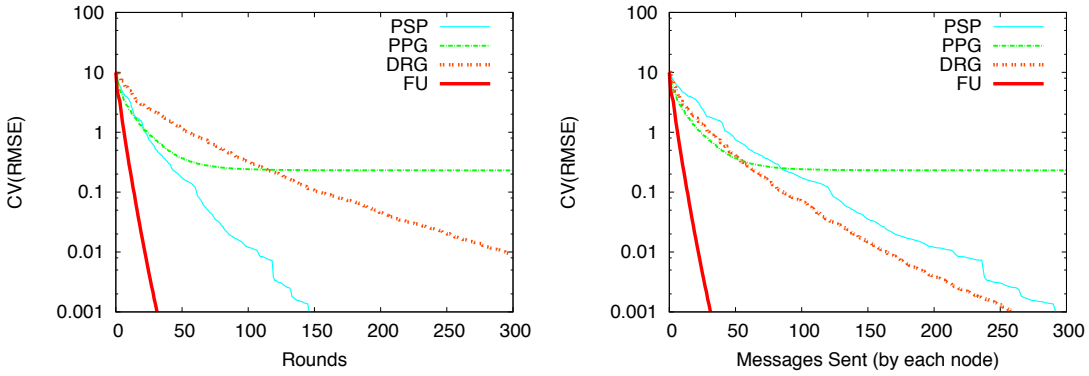


(e) Convergence speed ($n = 10000$)



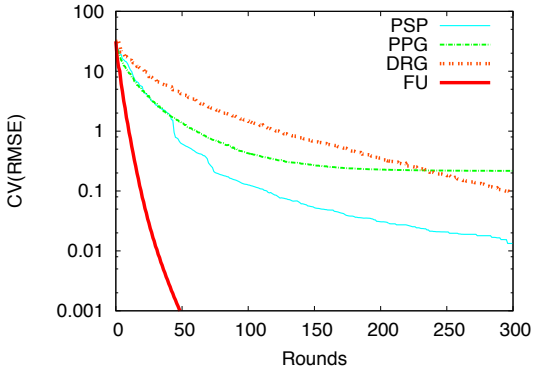
(f) Message load ($n = 10000$)

Figure 7.2: Comparison of *Flow Updating* against other averaging algorithms, on random networks with $d \approx 10$ and different sizes n (i.e., 100, 1000, 10000).

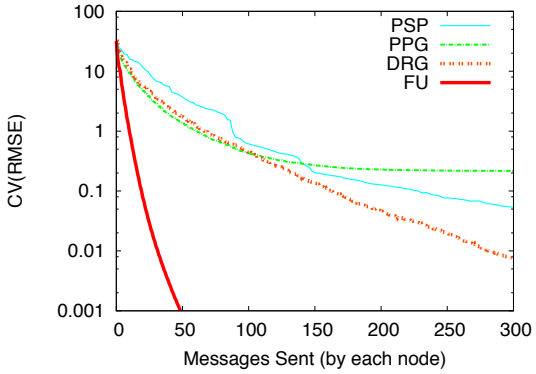


(a) Convergence speed ($n = 100$)

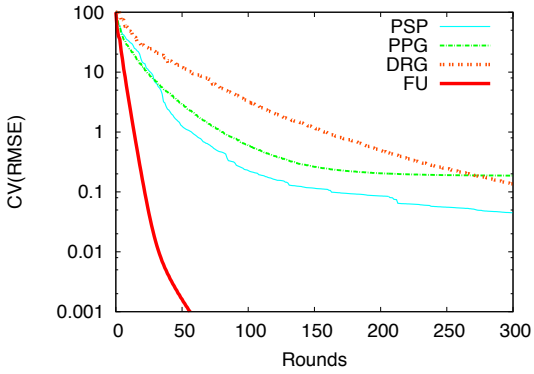
(b) Message load ($n = 100$)



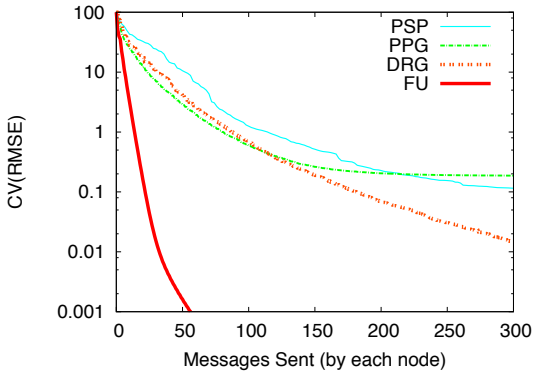
(c) Convergence speed ($n = 1000$)



(d) Message load ($n = 1000$)

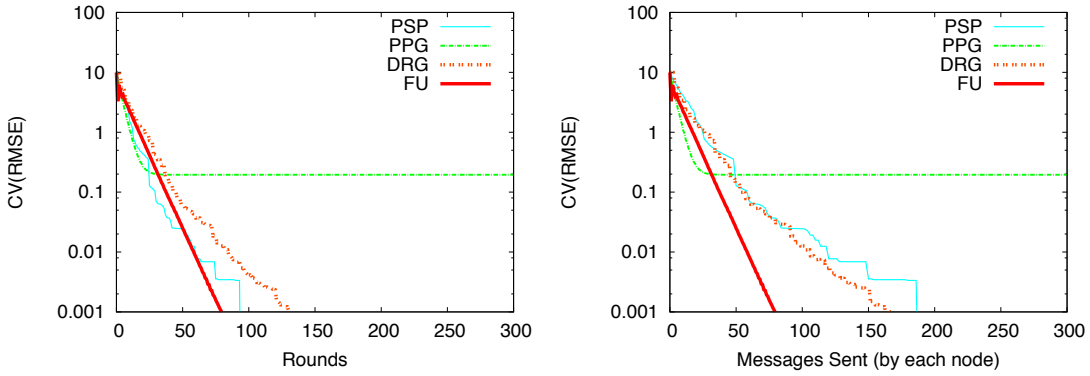


(e) Convergence speed ($n = 10000$)



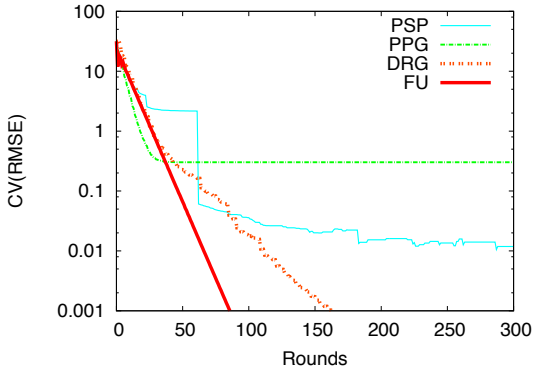
(f) Message load ($n = 10000$)

Figure 7.3: Comparison of *Flow Updating* against other averaging algorithms, on attach networks with $d \approx 3$ and different sizes n (i.e., 100, 1000, 10000).

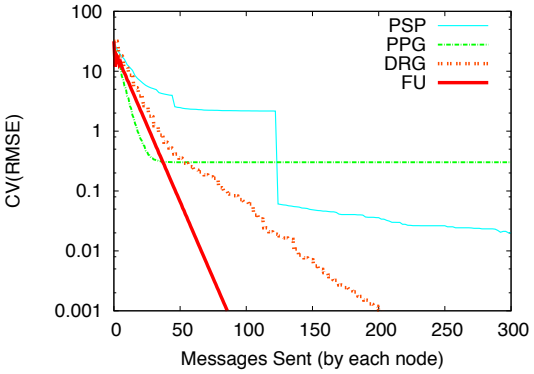


(a) Convergence speed ($n = 100$)

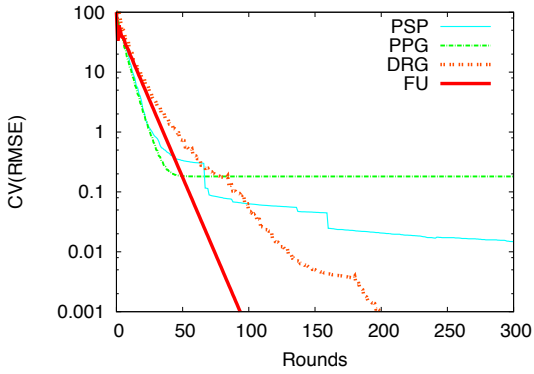
(b) Message load ($n = 100$)



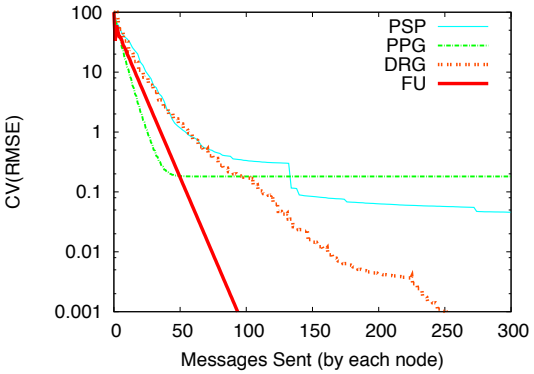
(c) Convergence speed ($n = 1000$)



(d) Message load ($n = 1000$)

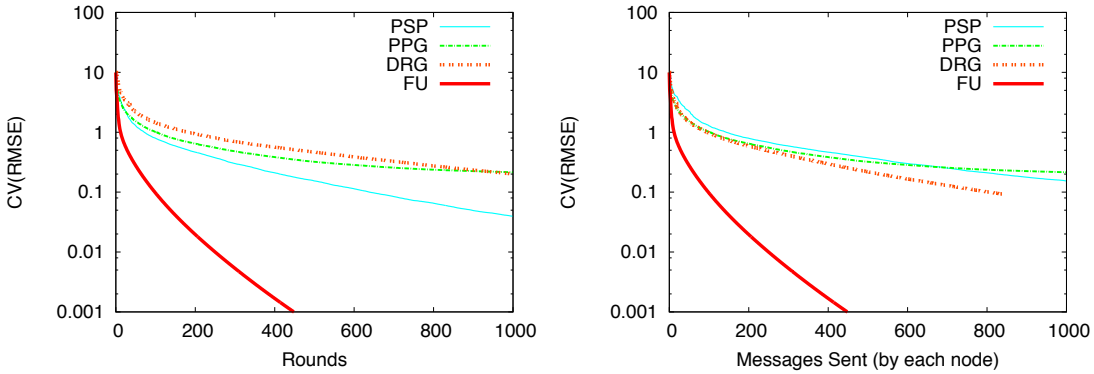


(e) Convergence speed ($n = 10000$)



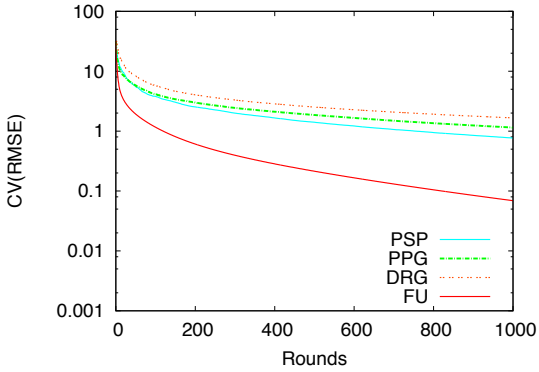
(f) Message load ($n = 10000$)

Figure 7.4: Comparison of *Flow Updating* against other averaging algorithms, on attach networks with $d \approx 10$ and different sizes n (i.e., 100, 1000, 10000).

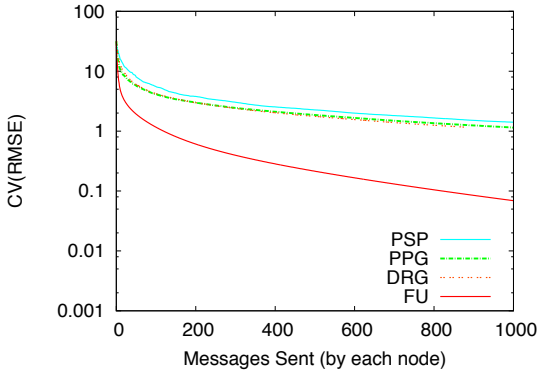


(a) Convergence speed ($n = 100$)

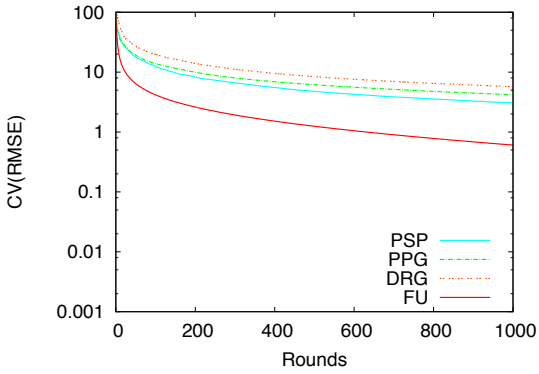
(b) Message load ($n = 100$)



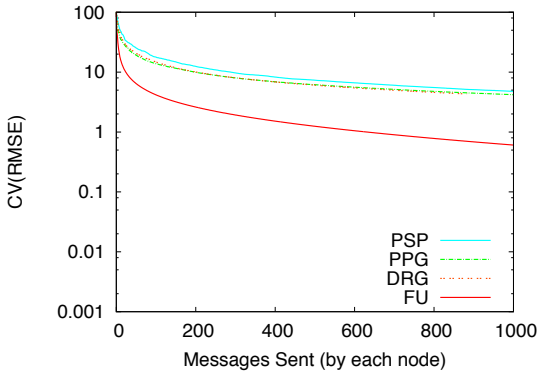
(c) Convergence speed ($n = 1000$)



(d) Message load($n = 1000$)

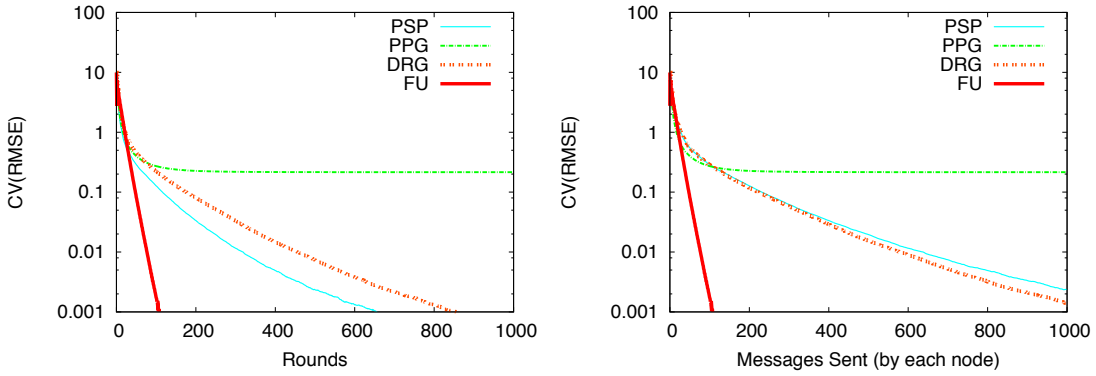


(e) Convergence speed ($n = 10000$)



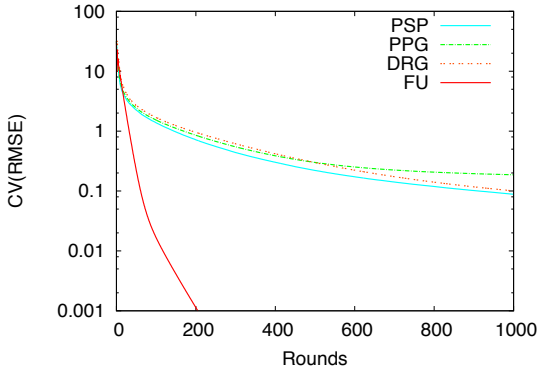
(f) Message load ($n = 10000$)

Figure 7.5: Comparison of *Flow Updating* against other averaging algorithms, on 2D/mesh networks with $d \approx 3$ and different sizes n (i.e., 100, 1000, 10000).

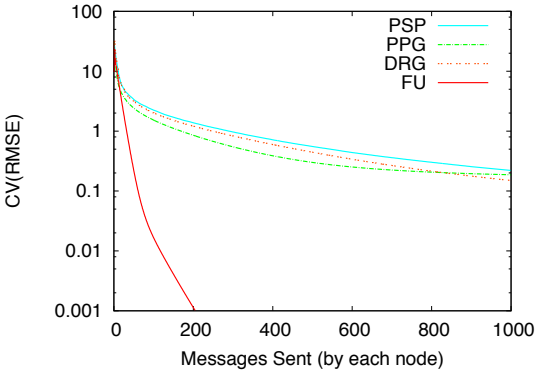


(a) Convergence speed ($n = 100$)

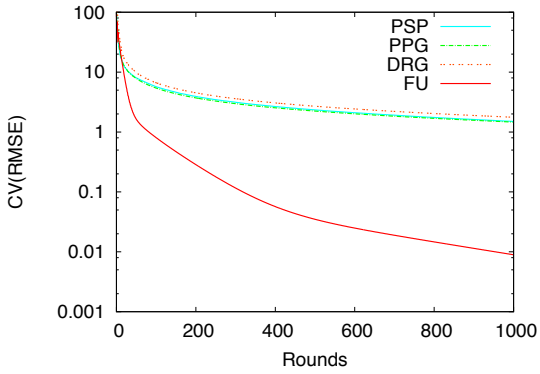
(b) Message load ($n = 100$)



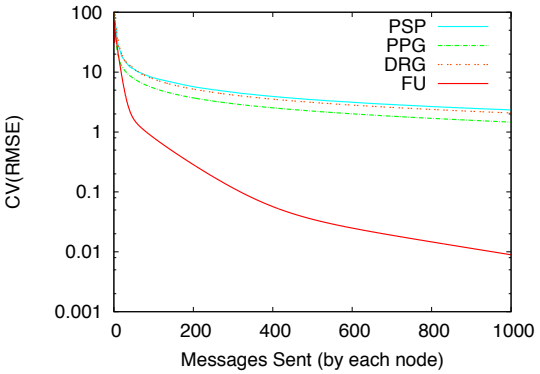
(c) Convergence speed ($n = 1000$)



(d) Message load ($n = 1000$)



(e) Convergence speed ($n = 10000$)



(f) Message load ($n = 10000$)

Figure 7.6: Comparison of *Flow Updating* against other averaging algorithms, on 2D/mesh networks with $d \approx 10$ and different sizes n (i.e., 100, 1000, 10000).

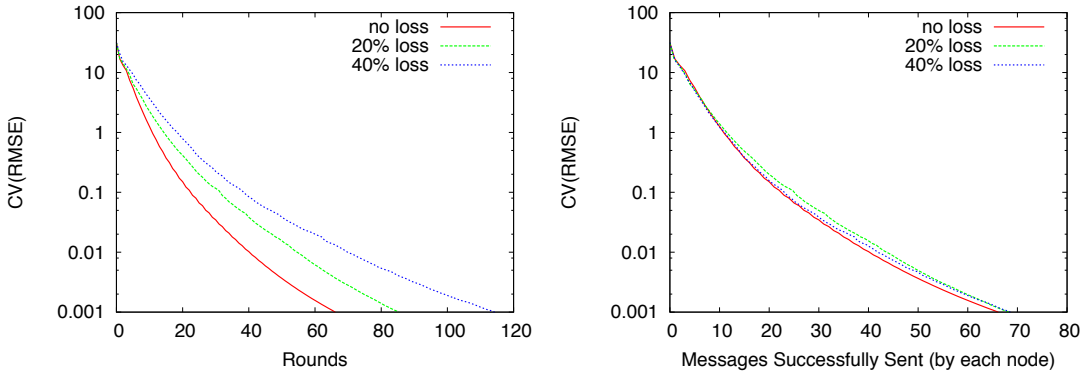
with a higher connection degree (i.e., $d \approx 10$), unlike the other compared algorithms which exhibit the opposite behavior (i.e., better performance for the higher connection degree). In general, the results obtained for attach networks are similar to the ones observed on random topologies. Nonetheless, a distinct behavior is perceived on 2D/mesh networks, and the performance degradation of FU for the higher connection degree is no longer verified. In fact, the performance of FU increases for $d \approx 10$. In this type of networks, FU considerably outperforms the other techniques, although the overall performance of all algorithms is significantly worst when compared to random and attach networks.

7.3 Fault-Tolerance

Here, the robustness (i.e., fault-tolerance) of *Flow Updating* is evaluated. Different types of failures can occur in a distributed system, for instance at the process level (e.g., when the battery of a device runs out of energy in a mobile system) or the communication level (e.g., due to interferences in a radio transmission, or message collision in WSN). More specifically, two types of faults are studied: node crash and message loss. Byzantine faults are not considered.

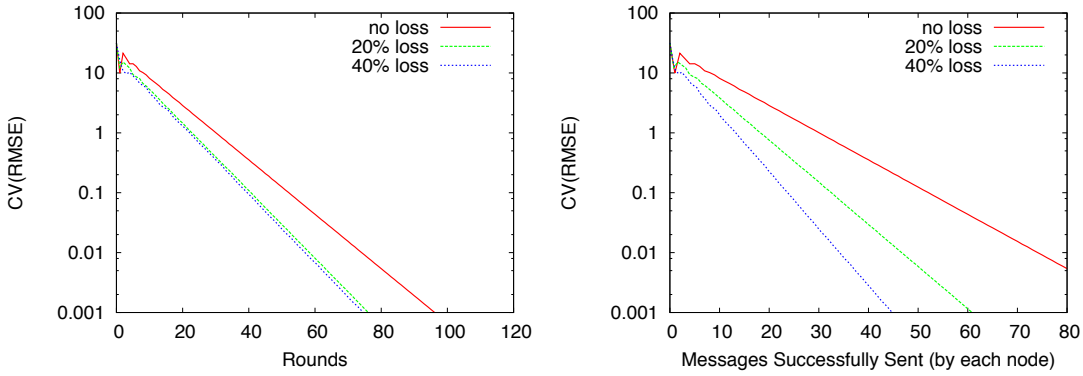
Node crash refers to the permanent failure of a node at an arbitrary time – *crash-stop* model. If a node crashes, it will no longer receive nor send messages, and will be considered as permanently unavailable from that time on. In practice, this kind of fault is equivalent to the unadvertised departure of a node. Since in dynamic setting it is assumed that a node can silently leave the network (without any notification, as notification messages may be lost), the impact of a node crash is equivalent to a node departure. Therefore, node crash will be evaluated indistinctively from nodes leaving the network in Section 7.5.1, where churn is considered.

Message loss correspond to the loss of communication data, due to a temporary link failure. In order to evaluate the impact of message loss, it is considered that each message sent in each round can be lost according to a predefined probability l. In particular, three levels of message loss were considered, 0% (no loss), 20% ($l = 0.2$) and 40% ($l = 0.4$). These values are compared in random and 2D/mesh networks of size $n = 1000$ and average degree $d \approx 3$ and $d \approx 10$. Note that classic averaging algorithms are not able to operate in faulty scenarios (without additional extensions), for the reasons stated in Section 4.1; therefore, only FU is evaluated on these settings.



(a) Convergence speed (b) Message load

Figure 7.7: *Flow Updating* with message loss on random networks ($n = 1000$, $d \approx 3$).

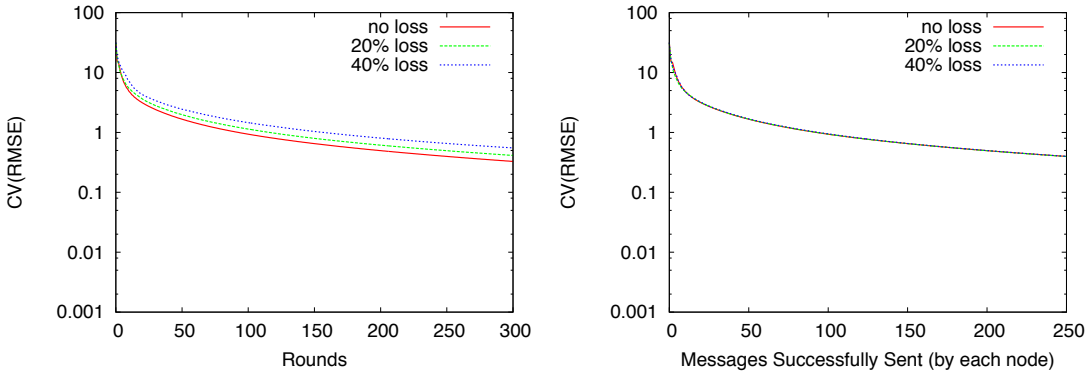


(a) Convergence speed (b) Message load

Figure 7.8: *Flow Updating* with message loss on random networks ($n = 1000$, $d \approx 10$).

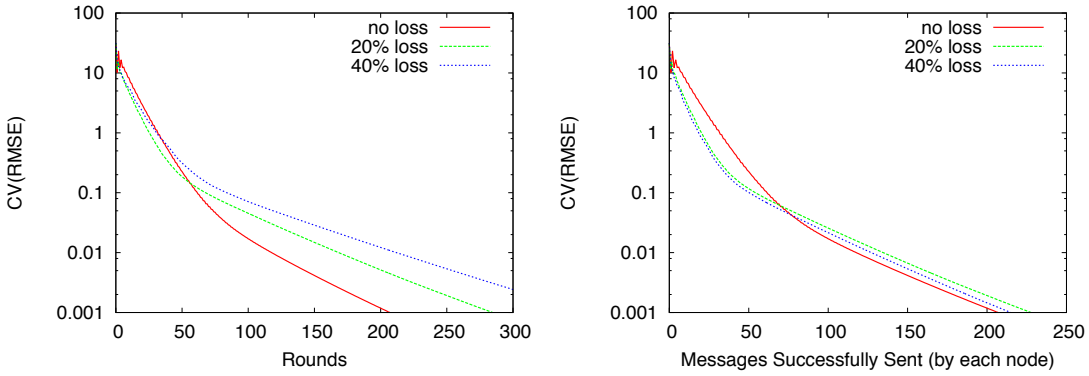
The obtained results are depicted by Figures 7.7–7.10. As can be observed, in general, message loss reduces the performance of FU (proportionally to the fault rate), but without preventing the convergence of the algorithm. This results are in accordance with the claims made in Sections 5.2.2.1 and 5.2.3.1. Namely, a similar quantity of successfully delivered messages is required to reach the same accuracy, independently from the amount of message loss, suggesting that all successful transmissions contribute to the convergence of FU (and almost no extra messages are needed to recover from the effect of a loss, besides replacing the lost message).

Curiously, in some situations the algorithm even benefits from message loss, increasing its convergence speed (e.g., message loss in Figure 7.8). It was found out that it is possible to increase the convergence speed of FU by "deactivating" some



(a) Convergence speed (b) Message load

Figure 7.9: *Flow Updating* with message loss on 2D/mesh networks ($n = 1000$, $d \approx 3$).



(a) Convergence speed (b) Message load

Figure 7.10: *Flow Updating* with message loss on 2D/mesh networks ($n = 1000$, $d \approx 10$).

communication links. This deactivation also provides a considerable reduction on the number of messages required to reach a given accuracy level. In some cases, message loss reproduces this effect, which explains this counterintuitive favorable results. The exploration of mechanisms to control this feature and the study of this additional source of convergence speedups was left for future work, although an initial heuristic has already been introduced in [73].

Moreover, it is interesting to observe that even under the occurrence of high amounts of message loss FU can still outperform classical algorithms operating under no message loss. For instance, comparing the results of Figures 7.1(c)–7.1(d) against Figure 7.7 for the random network scenario, and the results of Figures 7.6(c)–7.6(d) versus Figure 7.10 for 2D/mesh network, even with a substantial amount of message loss

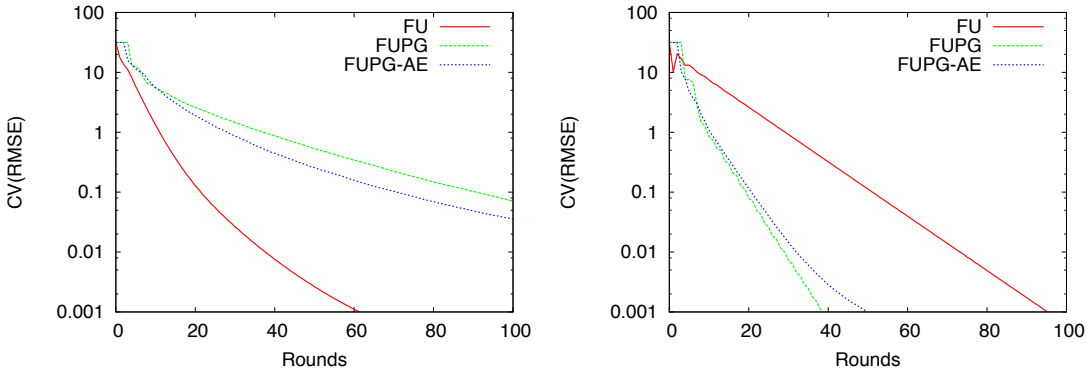
(40%) during the execution of FU, it outperforms the other algorithms without faults.

7.4 Flow Updating with Preferential Grouping

In this section, the performance of FU is compared with the variation defined in Section 5.3.1, i.e. Flow Updating with Preferential Grouping (FUPG). In particular, the two proposed group formation heuristics are evaluated. The version using the first heuristic (detailed by Algorithm 5) will be identified by the acronym FUPG, and the one applying the heuristic with average expectation (detailed by Algorithm 6) will be designated FUPG-AE.

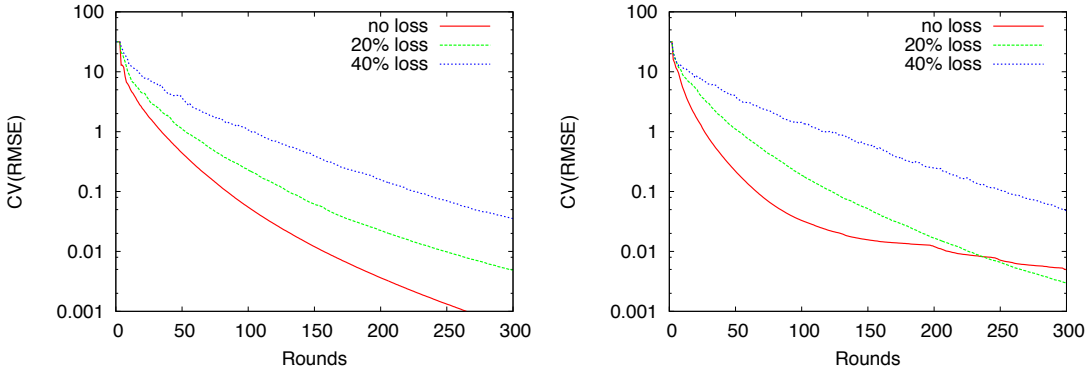
First, the performances of the algorithms are compared in fault-free scenarios, executing over random and 2D/mesh topologies. The networks have the same number of nodes $n = 1000$, and two different average degrees are considered, namely $d \approx 3$ and $d \approx 10$. The results obtained on random networks (Figure 7.11) reveal that FUPG and FUPG-AE outperform FU on networks with a higher connection degree (i.e., $d \approx 10$). In particular, unlike FU, we notice that the performance of FUPG and FUPG-AE is not affected by the increase of the connection degree. In fact, the converge speed of FUPG and FUPG-AE increases. On the other hand, FU outperforms FUPG and FUPG-AE on 2D/Mesh networks (Figure 7.14).

Finally, FUPG and FUPG-AE are evaluated in faulty simulation scenarios, with message loss. The same previous network settings are considered (i.e., random and 2D/mesh networks; $n = 1000$; $d \approx 3$ and $d \approx 10$), but now each sent message can be lost according to a given probability. The execution of the algorithms with different amounts of message loss are compared, namely with 0% (no loss), 20%, and 40%. The results are depicted by Figures 7.12, 7.13, 7.15, and 7.16. It is possible to observe that message loss affects the performance of FUPG and FUPG-AE, delaying the convergence, like in FU. However, comparing the results with the ones obtained for FU in Section 7.3, we can verify that message loss have a greater impact in FUPG and FUPG-AE than in FU. Therefore, attending to the overall better performance of FU, it seems the best choice to apply in realistic environments where messages are likely to be lost.



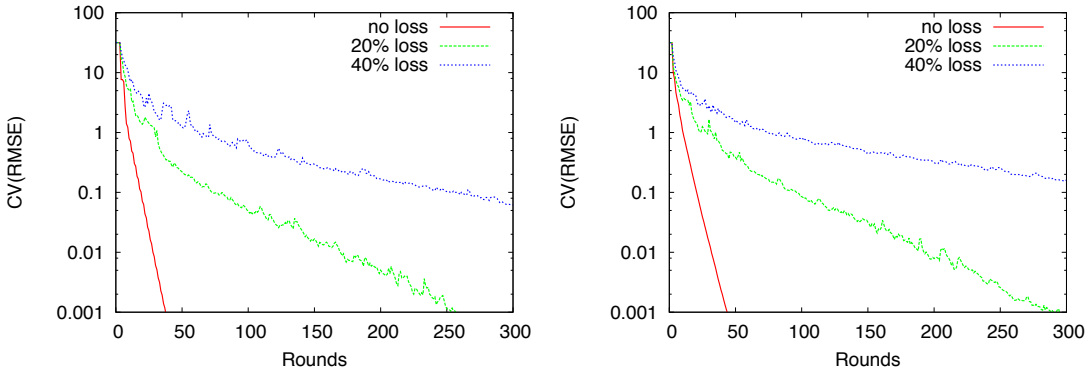
(a) Convergence speed ($d \approx 3$) (b) Convergence speed ($d \approx 10$)

Figure 7.11: Comparison of *Flow Updating* against its variations, on random networks with size $n = 1000$ and different average connection degrees (i.e., 3 and 10).



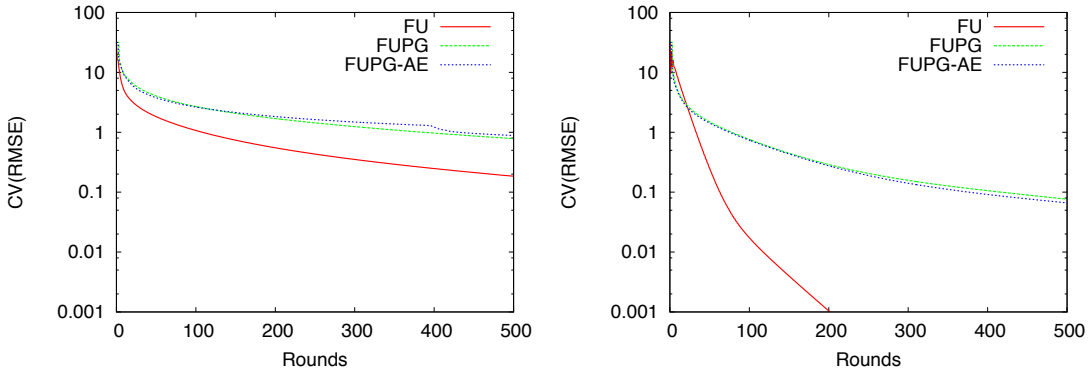
(a) FUPG (b) FUPG-AE

Figure 7.12: Variations of *Flow Updating* with loss – random networks ($n = 1000$; $d \approx 3$).



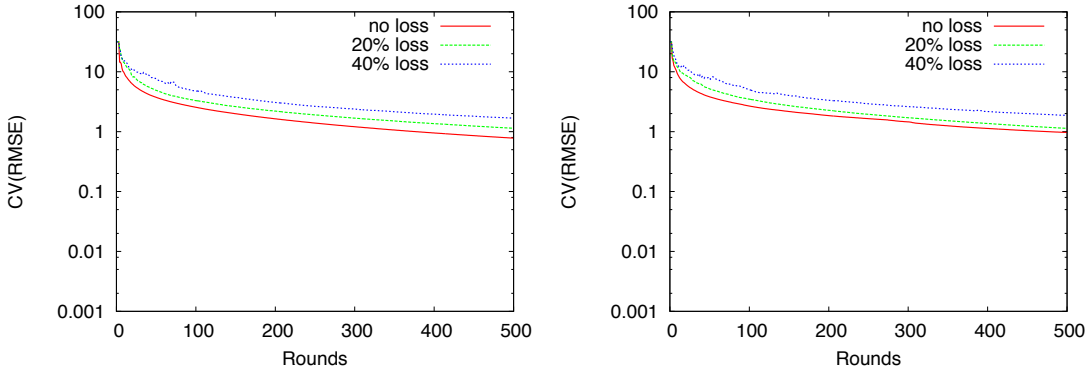
(a) FUPG (b) FUPG-AE

Figure 7.13: Variations of *Flow Updating* with loss – random networks ($n = 1000$; $d \approx 10$).



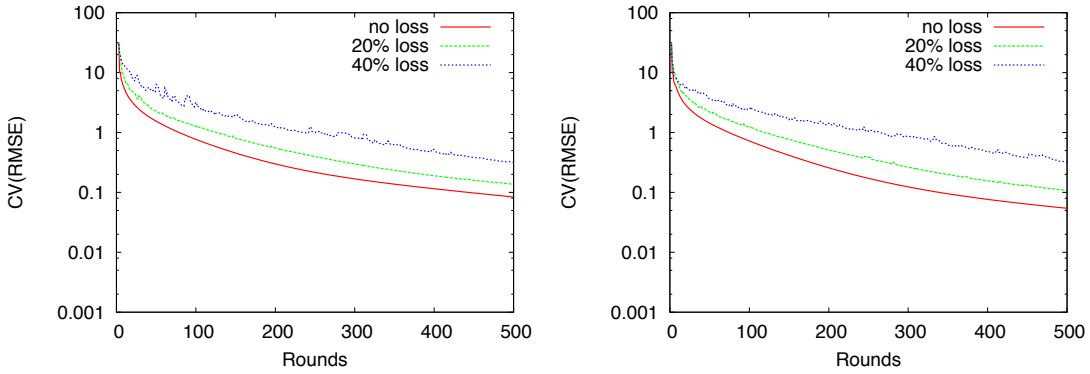
(a) Convergence speed ($d \approx 3$) (b) Convergence speed ($d \approx 10$)

Figure 7.14: Comparison of *Flow Updating* against its variations, on 2D/mesh networks with size $n = 1000$ and different average connection degrees (i.e., 3 and 10).



(a) FUPG (b) FUPG-AE

Figure 7.15: Variations of *Flow Updating* with loss – 2D/mesh networks ($n = 1000$; $d \approx 3$).



(a) FUPG (b) FUPG-AE

Figure 7.16: Variations of *Flow Updating* with loss – 2D/mesh networks ($n = 1000$; $d \approx 10$).

7.5 Dynamism

This section provides experimental results of the execution of FU under demanding dynamic settings, with message loss. FU is also compared to an existing averaged based technique intended to be able to operate on dynamic networks, namely Push-Pull Gossiping (PPG) [68] and *Push-Pull Ordered Wait* (PPOW) [71]. PPOW is a fix of PPG that solves its atomicity problems as referred in Section 4.1.2.

Two network topologies were considered, i.e. random and 2D/mesh. All networks considered in every scenario start with the same size ($n = 1000$), and the same approximated average connection degree ($d \approx \log n$)[3]. The choice of d was influenced by [77], where it is stated that some node must have a degree $\Omega(\log n)$ in order to keep the network connected with constant probability, considering that all nodes fail with a probability of 0.5. In general, the used value was enough to avoid network partitioning for the simulated churn scenarios (e.g., failure of one quarter of the nodes).

Two different dynamic setting are evaluated separately in the next sections: churn and input values change.

7.5.1 Churn

Churn refers to the departure and arrival of nodes. It is assumed that nodes silently leave the network, without notifying any other node. Node crashes are also implicitly covered by this silent departure situation, which is equivalent in this case.

The evaluated churn scenario considers both drastic and continuous changes of the network membership. In particular, it successively applies the sudden departure (catastrophic crash) and arrival of 25% of the initial nodes, followed by a continuous arrival and departure of the same portion of nodes at a constant rate (10 nodes per round). Note that, for a matter of clarity, a stability period is introduced between each churn event. More specifically, for random networks a continuous churn rate of 10 nodes per round and a stability period of 50 rounds were used. However, in the case of 2D/mesh a bigger stability period (500 rounds) and slower churn rate (1 node per round) were considered, because the convergence speed on these kind of networks is much slower (see Section 7.2).

First, FU was compared to PPG and PPOW in the described dynamic scenario, without message loss and over random networks. PPG implements a restart mecha-

[3]Note that, the considered logarithmic is of natural base.

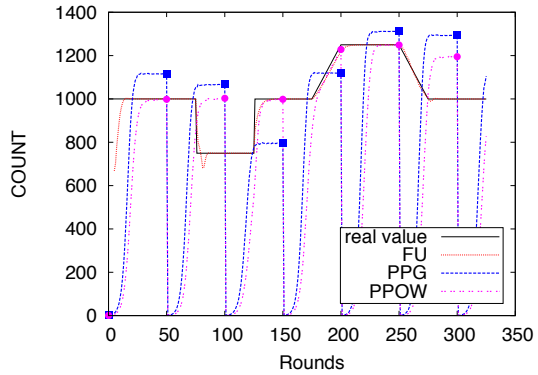
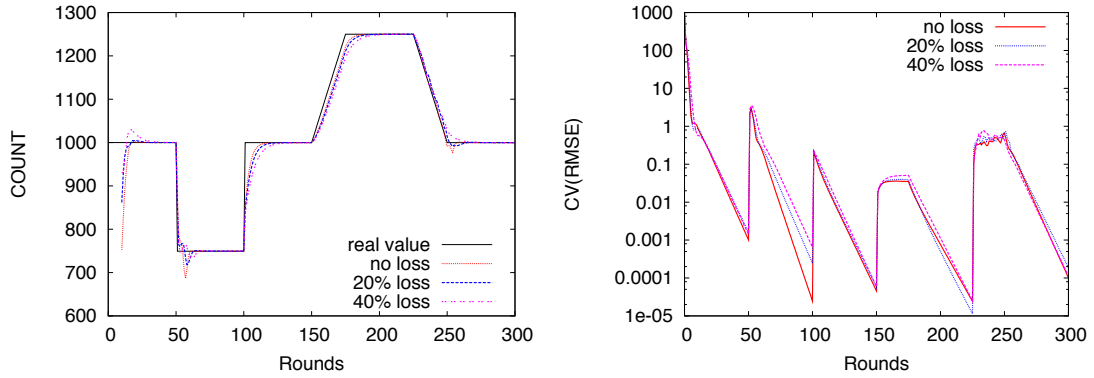


Figure 7.17: Comparison of FU in dynamic settings, with no message loss – random networks ($n = 1000$, $d \approx \log n$).

nism to cope with churn, starting a new instance of the algorithm after a predefined number of rounds (epoch), and not allowing new nodes to participate in the current running epoch. Similarly to PPG, PPOW was extended with a restart mechanism, but instead of delaying new nodes participation to the next epoch, joining nodes are allowed to immediately participate in the protocol. This modification was applied since it yielded more favorable results to PPOW in all performed experiments. An epoch length of 50 rounds is used for PPG and PPOW.

The obtained results are depicted by Figure 7.17. We can observe that an overestimate is produced by PPG due to its atomicity problems, even without network changes (e.g., between round 0 and 50), which is solved by PPOW that converges to the expected value. More importantly, these results expose the effect of the restart mechanism (in PPG and PPOW), which introduces an undesirable delay to respond to network change. Note that, this delay is also observed even if only the estimate at the end of each epoch are considered as valid (points at the end of each PPG and PPOW epoch, every 50 rounds). In the particular case of PPG, the delay is present in both node departure and arrival. However, in PPOW the response time to changes is reduced in the case of nodes arrival by allowing joining nodes to immediately participate in the current epoch.

The utilization of a restart mechanism introduces a trade-off between the response time to network changes and the accuracy of the push-pull algorithms, preventing them from following the network change with high accuracy. In contrast, FU is able to closely follow the network changes without requiring any restart mechanism. In this settings, FU clearly outperforms the other approaches (PPG and PPOW) which are



(a) Average estimate (b) Convergence Speed

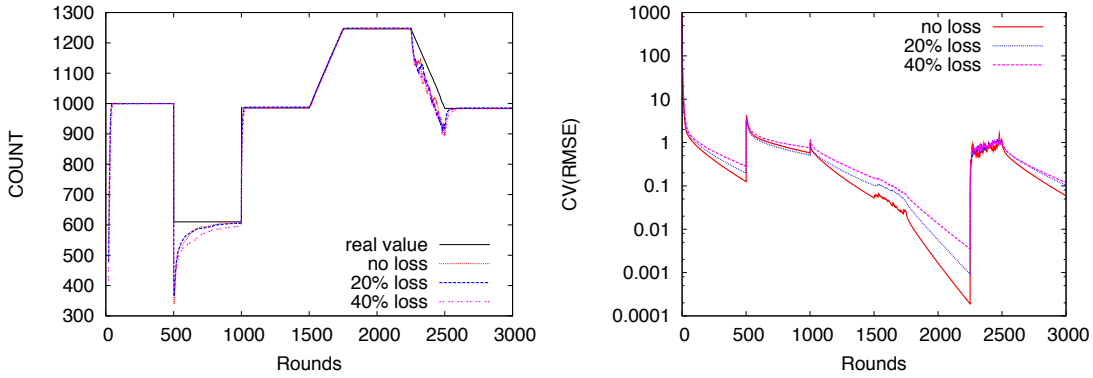
Figure 7.18: FU in dynamic settings, with message loss – random networks ($n = 1000$, $d \approx \log n$).

unable to adapt to network changes. Therefore, the remaining of the evaluation will focus exclusively on FU.

Now, the behavior of FU is evaluated on the same churn settings, but message loss are also taken into consideration (i.e., each received message can be lost according to a predefined probability). These simulation settings were applied to random and 2D/mesh networks. The results are respectively shown by Figures 7.18 and 7.19. Notice that, the CV(RMSE) is a metric that compares each individual estimate against the actual network size, as perceived by an external observer that can inspect the whole network in 0 rounds. Therefore, it is a very demanding metric to observe the global accuracy variation due to dynamism, since in any actual distributed algorithm nodes would require a delay proportional to diameter rounds before knowing the network size.

On random networks, the average of the estimates produced by FU closely follows the network changes, as shown by Figure 7.18(a). Moreover, it is observed (from Figure 7.18) that message loss, even in considerable amounts 20% and 40%, only slightly affects convergence speed and the ability of the algorithm to cope with churn.

The results from Figure 7.18(b) confirm the fast convergence of the algorithm during stable periods, and show expected accuracy decreases (increase of the CV(RMSE)) resulting from network changes. Brutal changes lead to momentary perturbations which are rapidly reduced, while continuous changes will provoke an accuracy reduction that persists during the continuous churn time period. In the particular case of random networks, for the considered churn rate (10 nodes per round), the arrival



(a) Average estimate (b) Convergence Speed

Figure 7.19: FU in dynamic settings, with message loss – 2D/mesh networks ($n = 1000$, $d \approx \log n$).

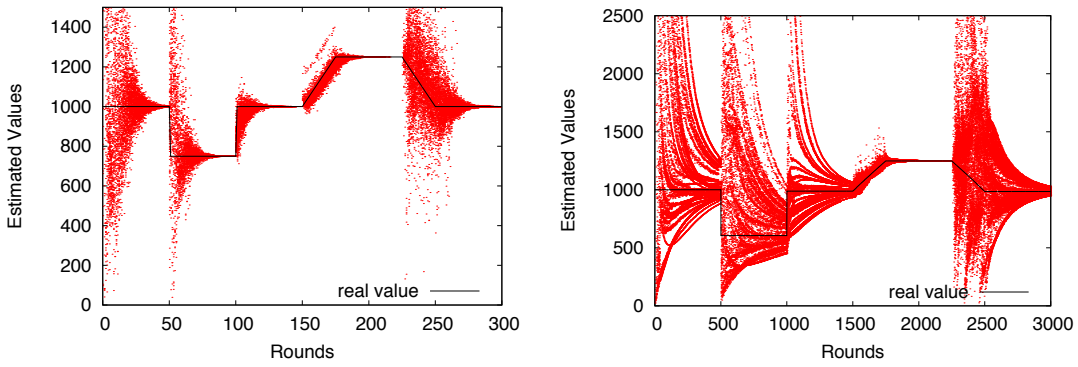
of nodes will increase the global error from less than 0.01% to about 3.5%, and node departures will increase it from less than 0.01% to about 50%. Node departure (or crashes) induce higher perturbations than node arrivals; in both cases the higher the number of nodes involved the bigger is the impact on node estimation accuracy.

On 2D/mesh networks, Figure 7.19, the behavior of FU is similar to the one reported for random networks, although a deeper contrast between the effect of node arrival and departure is observed. Namely, the perturbation introduced by a sudden (round 1000) or continuous (round 1500 to 1750) arrival of nodes is very small. On the contrary, node departure/crash have a greater impact in this type of networks.

Node departure/crash breaks the flows established between nodes, and can result in the removal of links that connect different network clusters, breaking the flows equilibrium in the whole network. This may lead to a global rearrangement of flows across the network, in order to reach a new equilibrium state. On the other hand, new nodes will simply provide new links (alternative paths), without breaking existing ones, which will lead to a smaller adjustment of the existing flows in order to converge to the new aggregation result (i.e., average).

The effect of churn on each node estimate is clearly depicted by Figure 7.20, which plots the individual estimates of all nodes over time, considering 20% of message loss[4]. Overall experimental results show that FU can provide accurate aggregation results in demanding dynamic and faulty networks. It allows all nodes to continuously adjust their estimates according to network changes, due to node arrival/departure and

[4]The graphs in a scenario without message loss is very similar to the case of 20% faults.



(a) Random network (b) 2D/Mesh network

Figure 7.20: Estimates distribution of FU in dynamic settings, with 20% of message loss.

crashes, quickly converging to the current network average, even with very high levels of message loss. However, until now, perfect failure detection has been assumed. Next, the application of practical failure detectors is evaluated, and we will see that FU will still behave nicely using conservative fault detection.

7.5.1.1 Fault Detection

Failure Detectors (FD) are important in the design of distributed system, abstracting the temporal uncertainties of realistic systems [19]. FD are unreliable and provide the information of processes that are suspected of having failed, making mistakes. Two main types of mistakes may occur: incorrect suspicions, when the FD incorrectly suspects a correct process; non suspicions, when a faulty process is not suspected by the FD. In this section, the impact of the application of realistic unreliable FD in the execution of FU on dynamic settings is evaluated.

Practical implementations of FD are commonly timeout-based [38]. Therefore, a simple timeout based implementation was considered, marking a node as suspected if no message is received from it after a predefined timeout value. The evaluation was carried out using the same succession of churn events of the previous simulations settings (i.e., sudden departure/arrival of a large amount of nodes, and continuous arrival/departure of a small number of nodes at a constant rate), and on random networks with the same setting (i.e., $n = 1000$ and $d \approx \log n$). The use of several FD with different timeout values was compared, ranging from 1 round (aggressive FD) to 4 rounds (conservative FD), and including a perfect FD as baseline. Three scenarios of

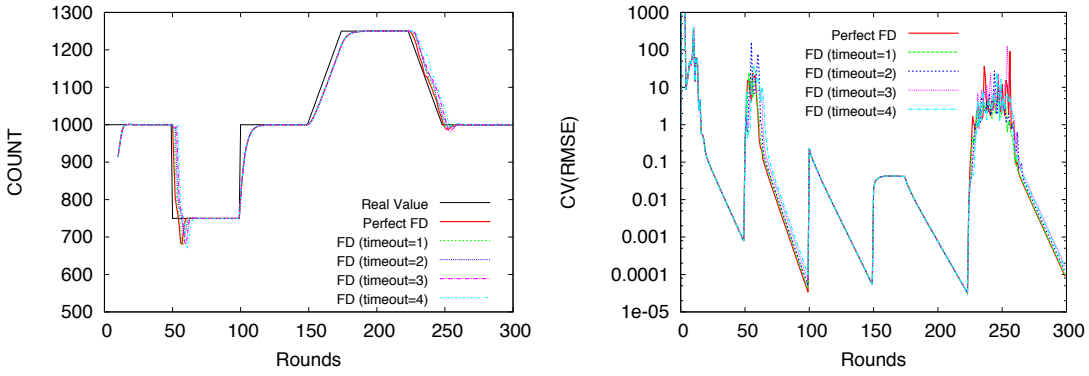
message loss were evaluated: no loss, 10% and 20% of message loss.

The results of Figure 7.21 show how the performance of FU is affected by FD with different timeout values, when subjected to churn and message loss, and Figure 7.22 depicts the measured number of mistakes made by each FD for each corresponding experimental setting. As expected nodes departure/crash lead to non suspicions by the FD, while message loss originate incorrect suspicions.

Each FD takes timeout rounds to detect the departure/crash of a node, never suspecting the leaving node during that time. Therefore, only after timeout rounds FU will be informed of the departure/crash of nodes, incurring on a delay proportional to the FD timeout to react to departures/crashes, as shown by Figure 7.21(a) and 7.21(b). Nonetheless, the impact of this delay is not very significant and FU is still able to closely adapt to changes.

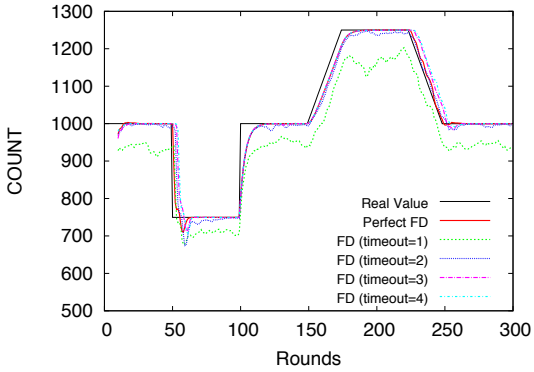
On the other hand, message loss can significantly impact the performance of FU when using aggressive FD. Message loss might lead to the incorrect suspicion of some nodes. In this case, FU will naively remove the flows of a correct process, and the whole system will start converging to a new (incorrect) average. Upon the reception of a message from an incorrectly suspected node, its flow will be immediately restored, and the convergence will be back on track toward the correct result. Although, since message loss occurs continuously over time, this situation might also occur recurrently, especially with aggressive FD (i.e., with a small timeout), introducing a constant perturbation on the execution of FU and impairing its convergence toward the correct result. As shown by Figures 7.21(c)–7.21(d) and 7.21(e)–7.21(f), the higher the amount of message loss the higher will be the impact on FU, especially when using a FD with a small timeout. This is because, FD with small timeout values only require a few consecutive message loss to incorrectly mark a node as suspected (e.g., only one message loss is enough for the FD with timeout 1), while FD with bigger timeouts will require a proportional amount of consecutive message losses before incorrectly suspecting a node, which is less likely to happen for moderate message loss rates (see Figures 7.22(c) and 7.22(e)). Therefore, it is more appropriate to use conservative FD.

The obtained results show that the selection of an appropriate FD is fundamental to ensure a good performance and accuracy of FU. Despite the additional delay introduced by conservative FD to react to network changes (i.e., nodes departure/crash), this kind of FD should be preferred. More importantly, it is fundamental to use a practical FD that minimizes the number of incorrect suspicions, in order to avoid an

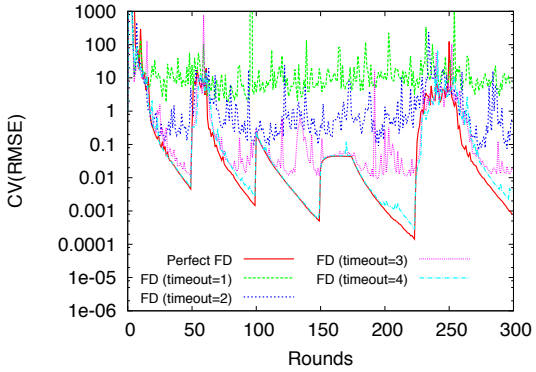


(a) Average estimate (no loss)

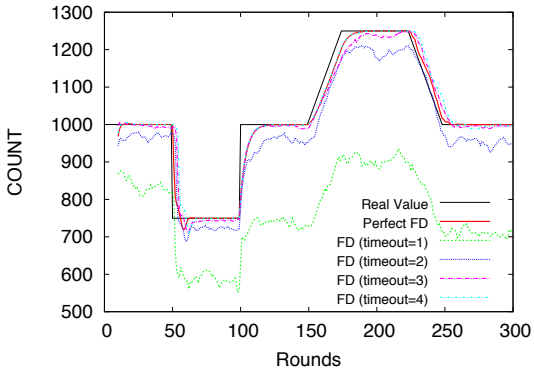
(b) Convergence speed (no loss)



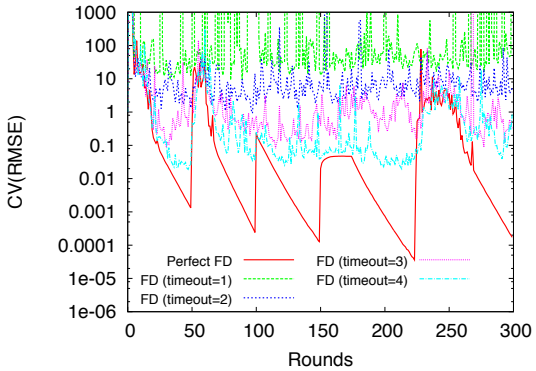
(c) Average estimate (10% loss)



(d) Convergence Speed (10% loss)

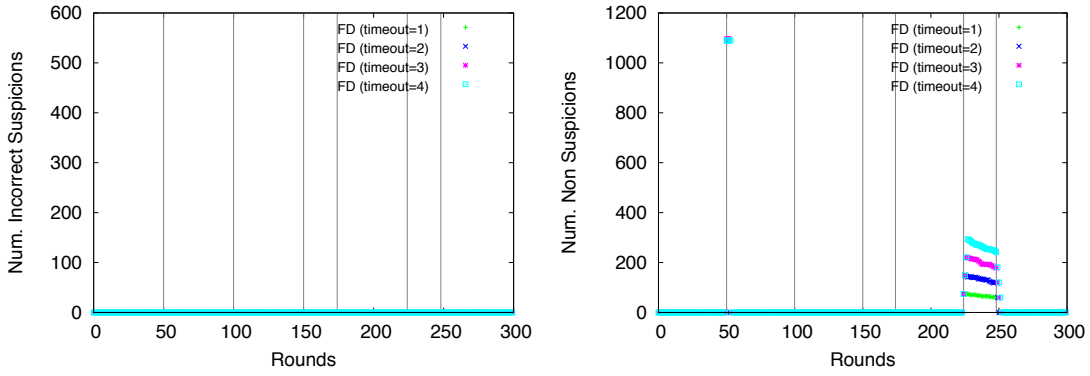


(e) Average estimate (20% loss)



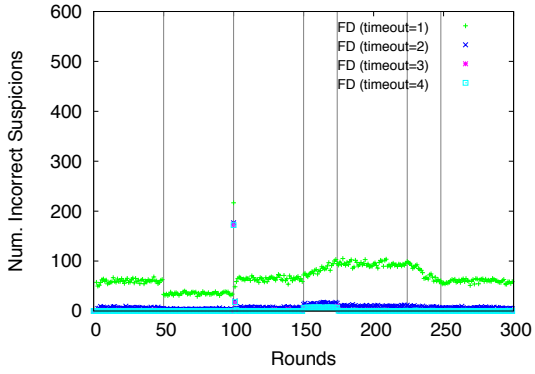
(f) Convergence Speed (20% loss)

Figure 7.21: Effect of FD on the execution of FU in dynamic settings with message loss – random networks ($n = 1000$, $d \approx \log n$).

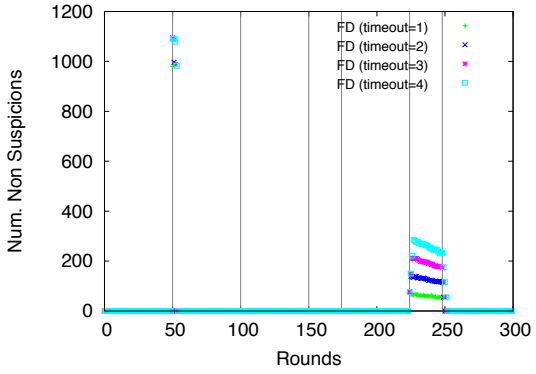


(a) Incorrect Suspicions (no loss)

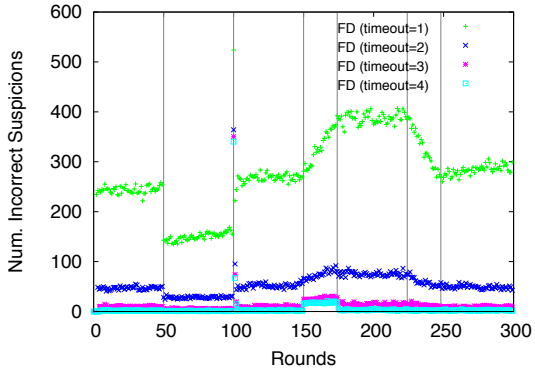
(b) Non Suspicions (no loss)



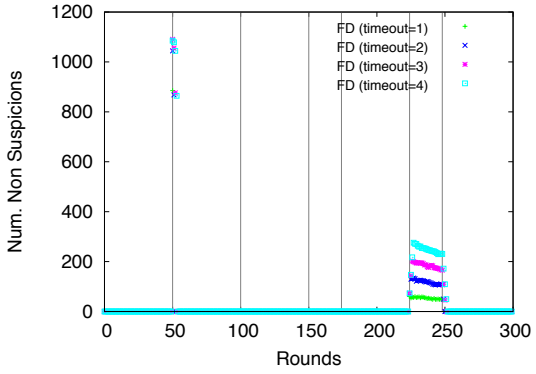
(c) Incorrect Suspicions (10% loss)



(d) Non Suspicions (10% loss)



(e) Incorrect Suspicions (20% loss)



(f) Non Suspicions (20% loss)

Figure 7.22: Mistakes of FD in dynamic settings with message loss – random networks ($n = 1000$, $d \approx \log n$).

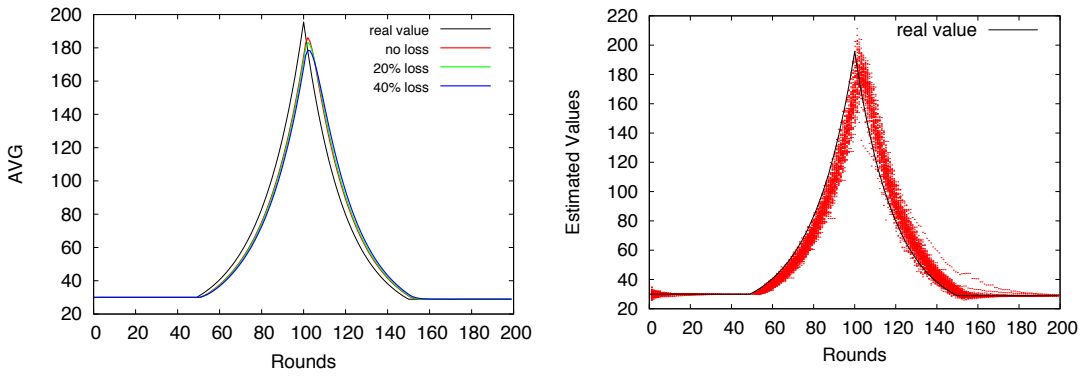
undesired performance degradation of FU. Note that, in general, this recommendation is valid for any network topology, as it is expected that fault detection will affect FU in the same way.

7.5.2 Input Values Change

Here, the behavior of FU is experimentally evaluated, when subjected to the dynamic change of the initial input values of the network nodes. For that purpose, a simple dynamic input value change scenario was defined, to compute the network average. Initially, each node starts with an input value chosen uniformly at random between 25 and 35, then after 50 rounds 50% of the nodes (randomly chosen) increase their input value 5% at each round, during 50 rounds, and then reduce their value by the same amount during another 50 rounds. These simulation settings intend to represent a possible variation of the temperature sensed by some nodes in an arbitrary monitoring environment. The execution of FU was compared considering different message loss amounts (i.e., 0%, 20% and 40%), over random networks ($n = 1000$ and $d \approx 3$).

The results of Figure 7.23(a) show that the average of the estimates produced by all nodes closely follow the change of the global average (with a small delay), independently from the amount of message loss. A more precise view of the estimate of all nodes over time is given by Figure 7.23(b), for a simulation with 20% of message loss. The results confirm that the estimates at all nodes closely follow the input changes, and that the difference between nodes estimates is small. The estimates graphs obtained in the scenarios with 0% and 40% of message loss (not shown) are very similar to the one depicted by Figure 7.23(b). Only a slightly variance on the difference between the estimates can be observed, being even smaller in the scenario without loss and a bit bigger with 40% of message loss.

In conclusion, FU is also able to self adapt to changes of the input values, closely tracking this kind of dynamic change with a small delay, even for large amount of message loss (i.e., 40%). Note that, in this case, no action is required by FU (i.e., no need to add/remove flows) in order to converge to the new result.



(a) Average estimate (b) Estimate with 20% of message loss

Figure 7.23: FU with input value changes, and message loss – random networks ($n = 1000$, $d \approx 3$).

7.6 Termination/Quiescence

The strategy previously proposed in Section 5.3.2 to implement termination/quiescence of FU is experimentally evaluated in this section. In particular, the number ci of consecutive iterations (rounds) that should elapse before FU stops sending messages (or reports the result) is analyzed for some specific scenarios.

In particular, FU was executed on random and 2D/mesh networks with the same number of nodes $n = 1000$, but with different average connection degrees (respectively, $d \approx 3$ and $d \approx 10$). Message loss is also taken into account, namely 10% and 20%. In terms of the parameters set for the termination/quiescence strategy: the same threshold ξ was set in all simulations, i.e. $\xi = 0.01$; different values of consecutive iterations ci (from 0 to 8) were compared for the same simulation settings. In addition, a special case was considered where the termination/quiescence strategy is applied but which never stops sending messages and only registers the entry/exit in a quiescent state, acting like the base FU without termination/quiescence. This special version is used as a baseline for comparison, and is identified by *send msg*.

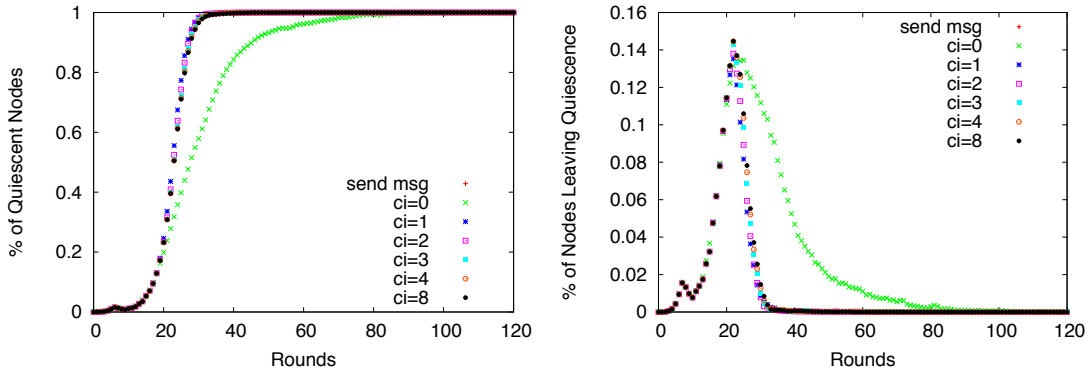
Figures 7.24, 7.25 and 7.26 show the results obtained on random networks, with and without message loss. In the scenario without message loss, we observe that if no rounds are waited ($ci = 0$) before a node enters in the quiescence state (i.e., stops sending messages), then it will take more time for all nodes to reach quiescence (Figure 7.24(a)) and some nodes will leave quiescence during more time (Figure 7.24(b)). However, it suffice to wait one round $ci = 1$ to observe a behavior in terms of nodes

entering/exiting quiescence similar to the baseline case (*send msg*).

In all scenarios, independently from the network and message loss rate, it is possible to observe an instability period, where a small fraction of nodes (at most 20%) continuously leave a previous quiescent state, until FU converges (to a value with an accuracy proportional to the defined threshold ξ) and a global quiescence state (i.e., termination) is reached. For example, in the random network scenario with no message loss, during at most 40 rounds for any $ci > 0$ (Figure 7.24(b)). The majority of this intermediary quiescence states will only last a few rounds (independently from the used ci), but rare occurrences of longer periods were also observed. For example, on random networks without loss more than 95% of these "false" quiescence periods do not last more than 5 rounds, as depicted by Figure 7.24(f) for $ci = 0$, but rare occurrences were found reaching close to 35 rounds. This situation will still happen even if the algorithm never stops sending messages, i.e., in the *send msg* version, although the maximum length of the observed "false" quiescence periods will be smaller (Figure 7.24(e) versus 7.24(f)).

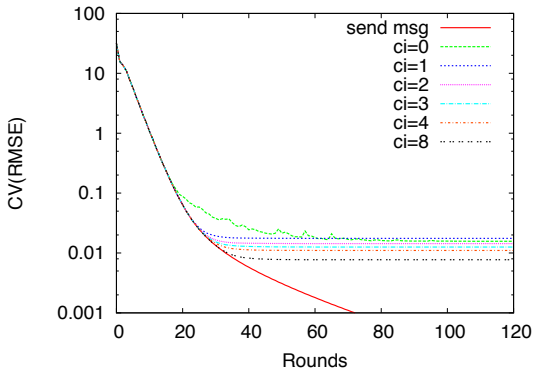
Figures 7.24(c) and 7.24(d) confirm that the use of $ci = 0$ is not the best choice for the random network scenario with no message loss. Namely, a better tradeoff in terms of message load is obtained for $ci = 1$, $ci = 2$, $ci = 3$ and $ci = 4$, when compared to $ci = 0$. The same can be observed for the scenario with 20% of message loss, as depicted in Figure 7.26. Moreover, in this particular case, for $ci = 0$ some nodes (i.e., less than 5%) are unable to reach quiescence. This may happen for nodes surrounded by quiescent neighbors that will not send messages that contribute to the computation of a new estimate. The occurrence of larger amounts of message loss seems to lead to this situation, if no extra round is waited before entering into a quiescent state. Curiously, this is not true for the settings with moderated message loss (i.e., 10%), as shown by Figures 7.25, where $ci = 0$ can also be considered a good choice. As expected, in general, using a higher value for ci will enable FU to reach a higher accuracy (for the same threshold ξ), since the algorithm executes during more time and exchanges more messages before reaching the final quiescence state.

The results obtained on 2D/mesh networks are depicted by Figures 7.27 and 7.28. In general, the results are consistent with the ones observed on random networks, confirming that the use of $ci = 0$ is not adequate in scenarios without message loss. However, some small differences can be observed on these experimental settings. Namely, we notice a more pronounced contrast between the quantities of nodes that leave qui-

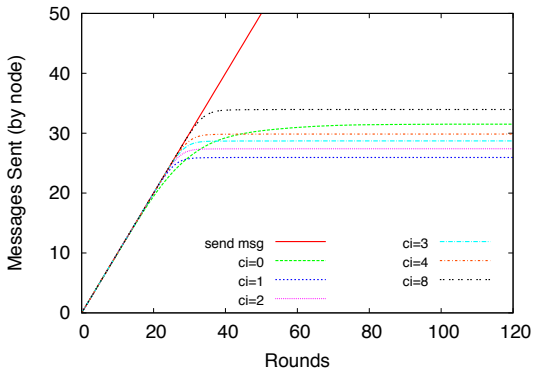


(a) Quiescent Nodes

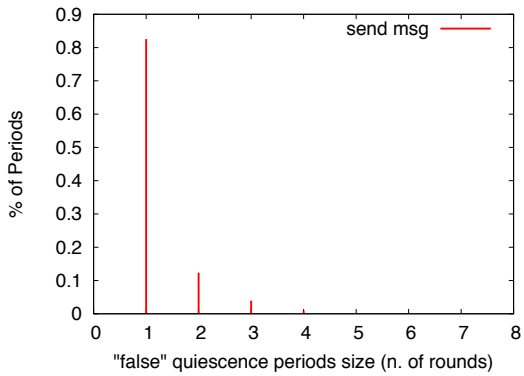
(b) Nodes Leaving Quiescence



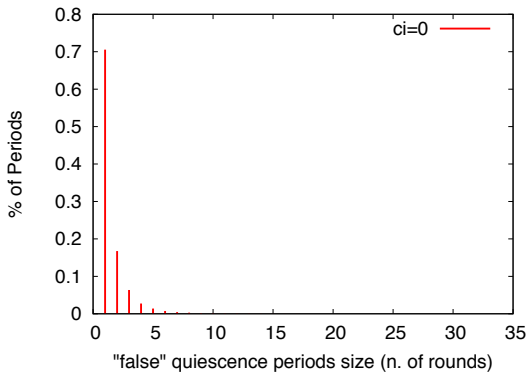
(c) Convergence/Accuracy



(d) Message load

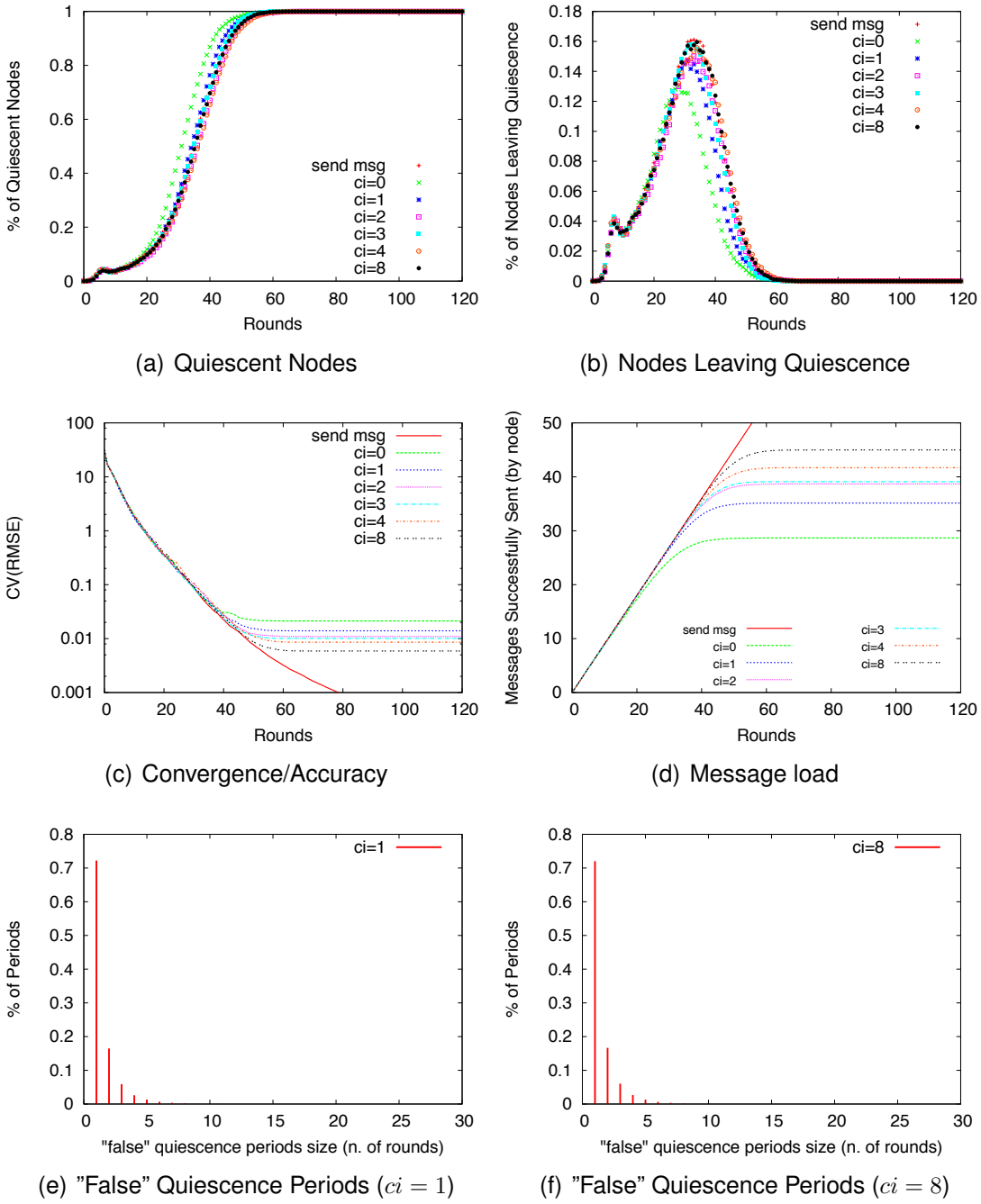


(e) ”False” Quiescence Periods (send msg)



(f) ”False” Quiescence Periods ($ci = 0$)

Figure 7.24: Quiescence with no message loss – random networks ($n = 1000$, $d \approx 3$).



(a) Quiescent Nodes

(b) Nodes Leaving Quiescence

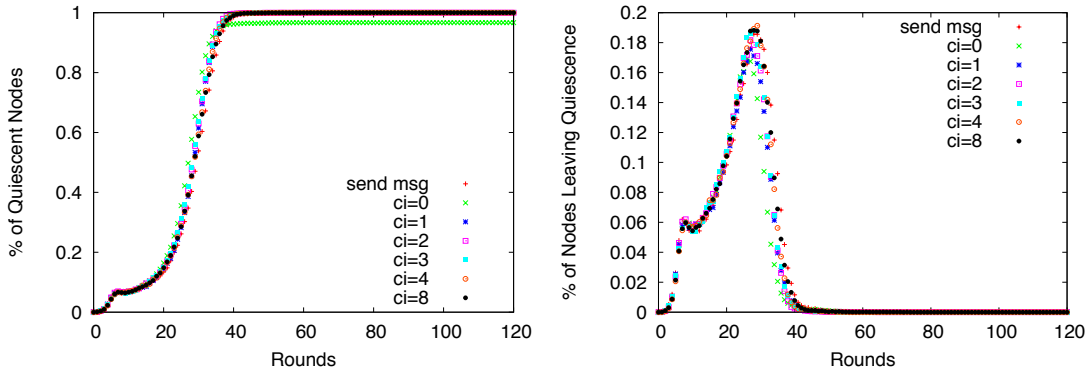
(c) Convergence/Accuracy

(d) Message load

(e) "False" Quiescence Periods ($ci = 1$)

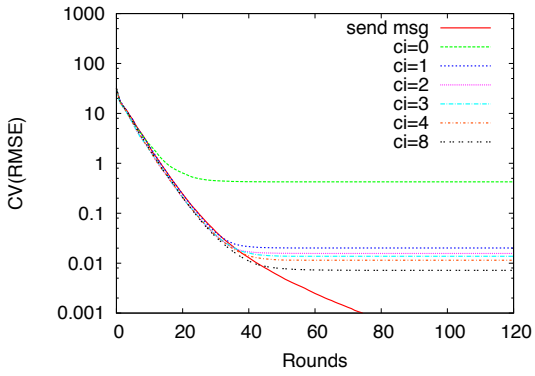
(f) "False" Quiescence Periods ($ci = 8$)

Figure 7.25: Quiescence with 10% of message loss – random networks ($n = 1000$, $d \approx 3$).

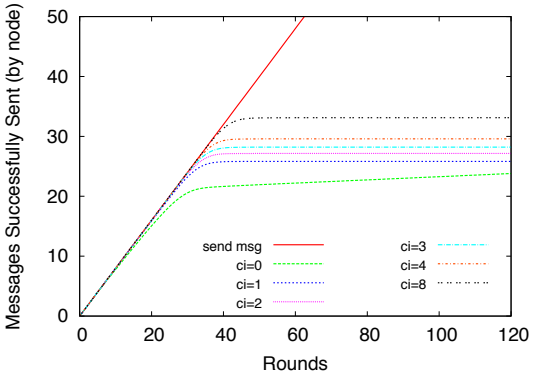


(a) Quiescent Nodes

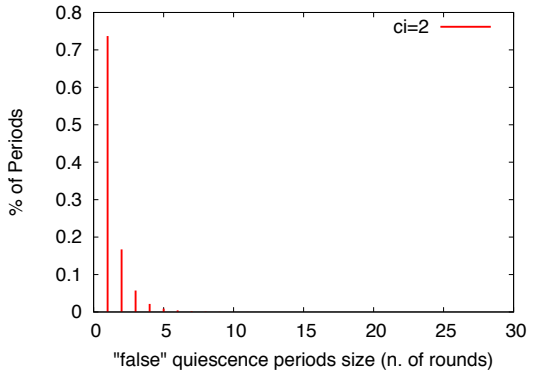
(b) Nodes Leaving Quiescence



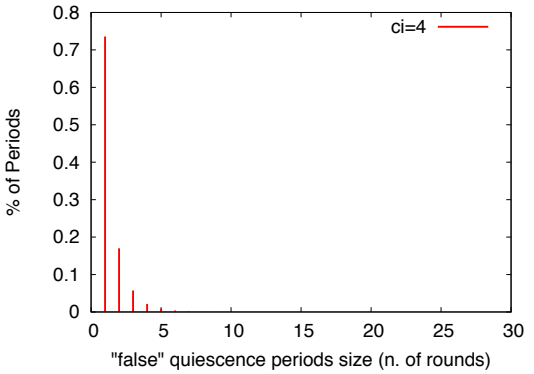
(c) Convergence/Accuracy



(d) Message load

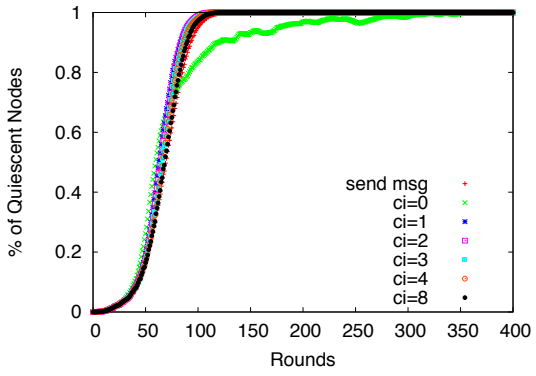


(e) ”False” Quiescence Periods ($ci = 2$)

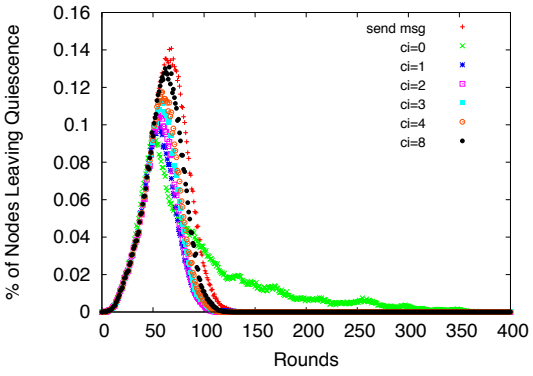


(f) ”False” Quiescence Periods ($ci = 4$)

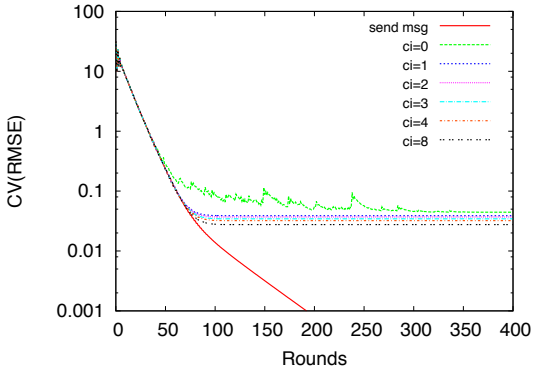
Figure 7.26: Quiescence with 20% of message loss – random networks ($n = 1000$, $d \approx 3$).



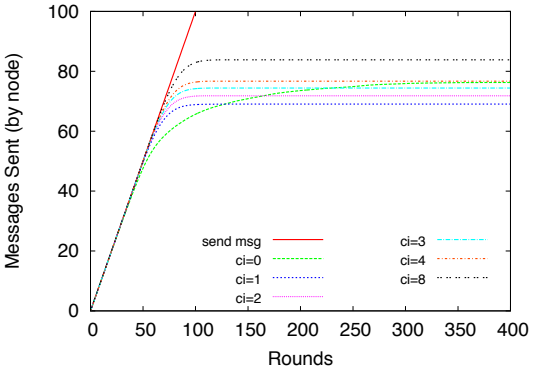
(a) Quiescent Nodes



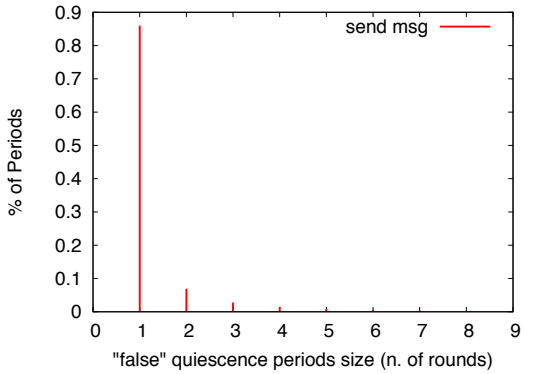
(b) Nodes Leaving Quiescence



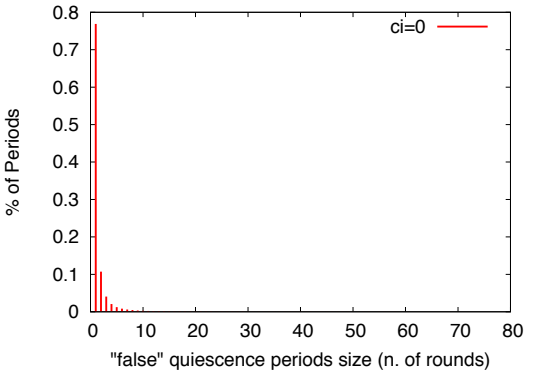
(c) Convergence/Accuracy



(d) Message load

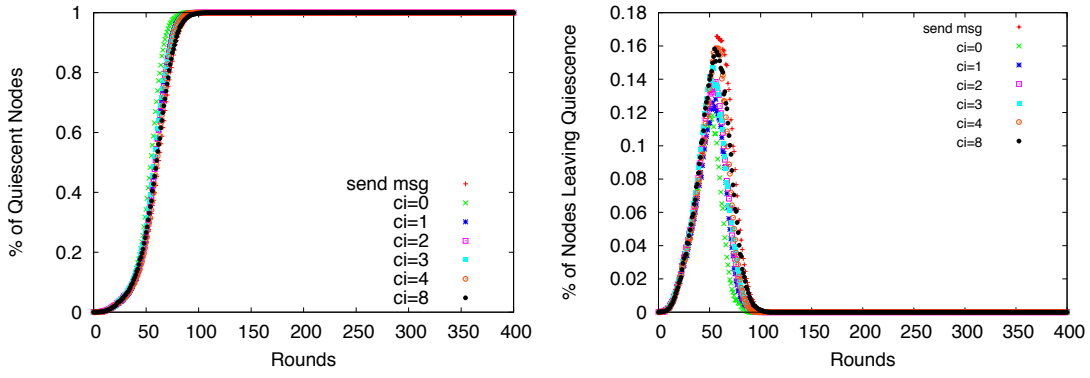


(e) ”False” Quiescence Periods (send msg)



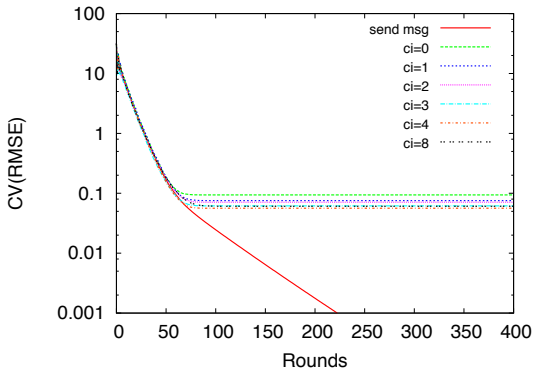
(f) ”False” Quiescence Periods ($ci = 0$)

Figure 7.27: Quiescence with no message loss – 2D/mesh networks ($n = 1000$, $d \approx 10$).

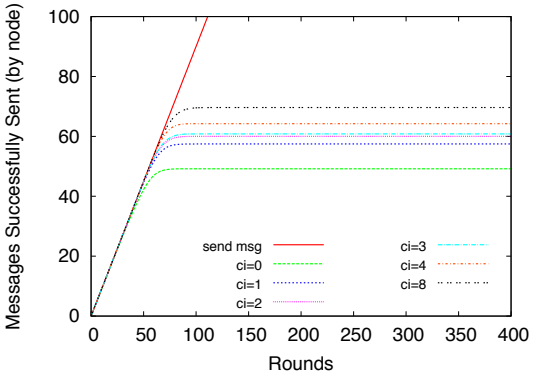


(a) Quiescent Nodes

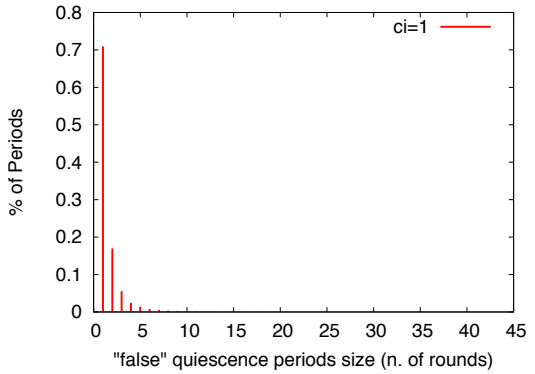
(b) Nodes Leaving Quiescence



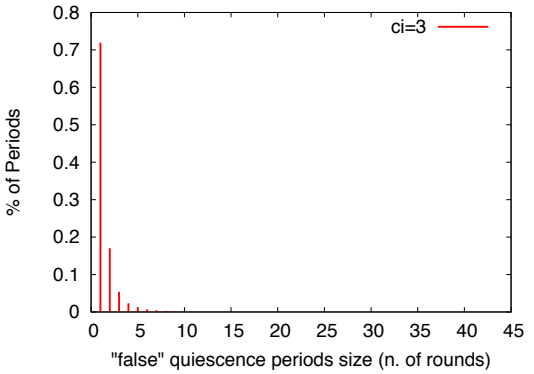
(c) Convergence/Accuracy



(d) Message load



(e) ”False” Quiescence Periods ($ci = 1$)



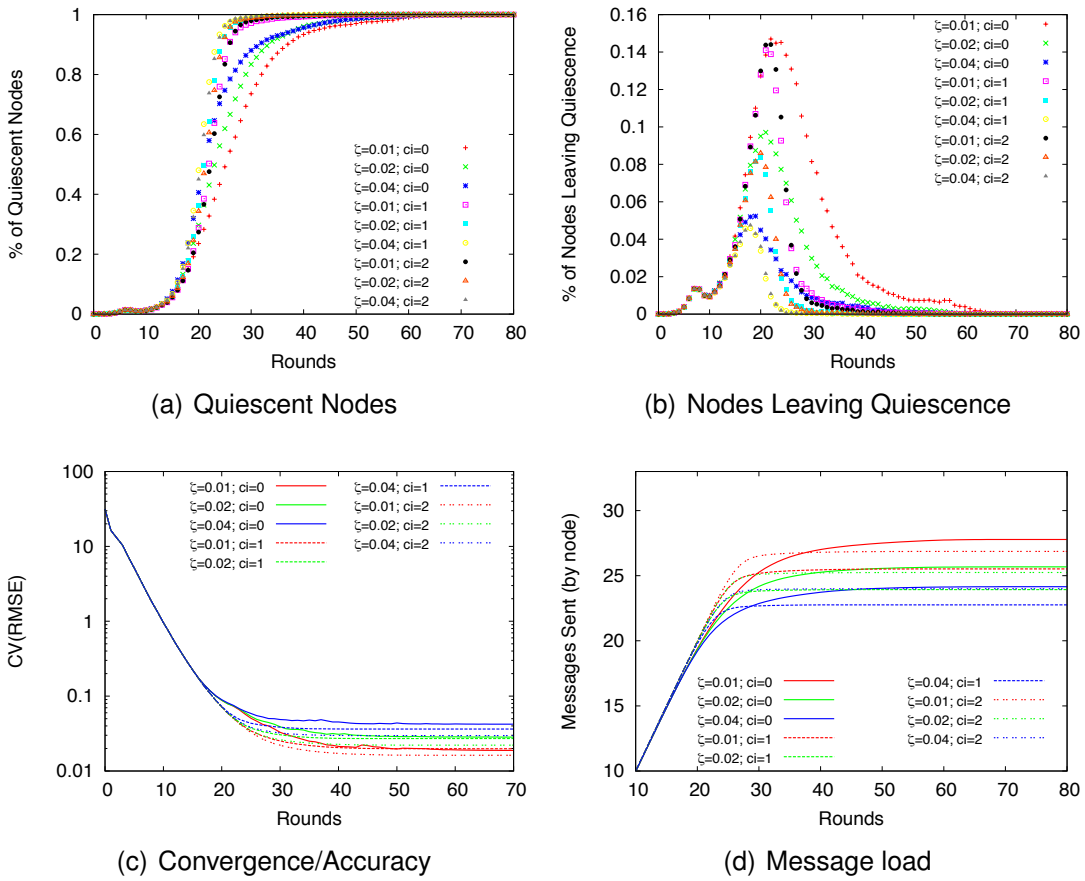
(f) ”False” Quiescence Periods ($ci = 3$)

Figure 7.28: Quiescence with 10% of message loss – 2D/mesh networks ($n = 1000$, $d \approx 10$).

escence (until convergence is reached) when using different waiting values ci, especially in the scenario without message loss (Figure 7.27(b)). Moreover, the "false" quiescence periods are longer for 2D/mesh when compared to random networks (e.g., Figure 7.24(f) versus Figure 7.27(f)). It is also noticeable that the accuracy reached is smaller than for random networks, using the same parameter values ($\xi = 0.01$ and ci). These dissimilarities can be explained by the worst performance of FU on 2D/mesh topologies when compared to random networks. For instance, in 2D/mesh networks the global quiescent state (i.e., termination) is only observed around round 100, while it is reached in general close to 60 rounds (even less) on random topologies. Regarding the number of consecutive iterations to wait before FU stops sending messages, the results on 2D/mesh networks (with and without message loss) suggest that a good tradeoff in terms of message load and accuracy is obtained for small ci values, but greater than 0 (e.g., $ci = 1$, $ci = 2$ and $ci = 3$).

In addition to the previous experimental simulations (i.e., assessing the use of different waiting times ci), the utilization of an additional threshold ζ to leave the quiescence state is also evaluated, with $\zeta > \xi$. The results obtained on random topologies ($n = 1000$, $d \approx 3$) with no messages loss are depicted by Figure 7.29. As expected, we observe that the use of higher ζ values reduces the amount of nodes that leave quiescence, as shown by Figure 7.29(b). A reduction of the number of message sent is also observed for higher thresholds ζ (see Figure 7.29(d)), however it is also associated to a decrease of the global accuracy reached by all nodes (see Figure 7.29(c)). Therefore, no apparent advantage seems to come from the use of a distinct threshold ζ to leave quiescence; using the same threshold ξ to enter/exit quiescence looks satisfactory. Similar results were obtained on scenarios with message loss.

In conclusion, considering the continuous execution of FU in *monitoring* mode, as described in Section 5.3.2, the experimental results show that the defined quiescence strategy is efficient independently from the network topology and even from message loss. It allows nodes to enter in a quiescent state when a certain level of accuracy is reached (depending from a defined threshold value ξ), avoiding the transmission of excessive and unnecessary amounts of messages beyond the desired precision. In general, a good trade-off between accuracy and message load was obtained using small ci values (e.g., $ci = 1$). Nevertheless, the obtained results are not enough to define a precise criteria for termination detection, due to the observed instability of nodes continuously leaving quiescence. One could use the maximum length of the "false"



(a) Quiescent Nodes (b) Nodes Leaving Quiescence

(c) Convergence/Accuracy (d) Message load

Figure 7.29: Use of different threshold values to leave quiescence – random networks ($n = 1000$, $d \approx 3$), with no message loss.

quiescence period to define a criteria for termination, although this value varies according to the network settings and may be affect by the occurrence of message loss. From the evaluated scenarios, staying more than 5 rounds for random networks and more than 20 for 2D/mesh in a quiescent state appears to be a sufficient to detect the termination of the majority of the nodes, although a few may terminate prematurely. The definition of a more precise criteria for termination detection is left for future work.

7.7 Asynchrony

In this section, the behavior of FU on asynchronous settings is evaluated, more specifically, taking into account the message latency distribution present in wide area net-

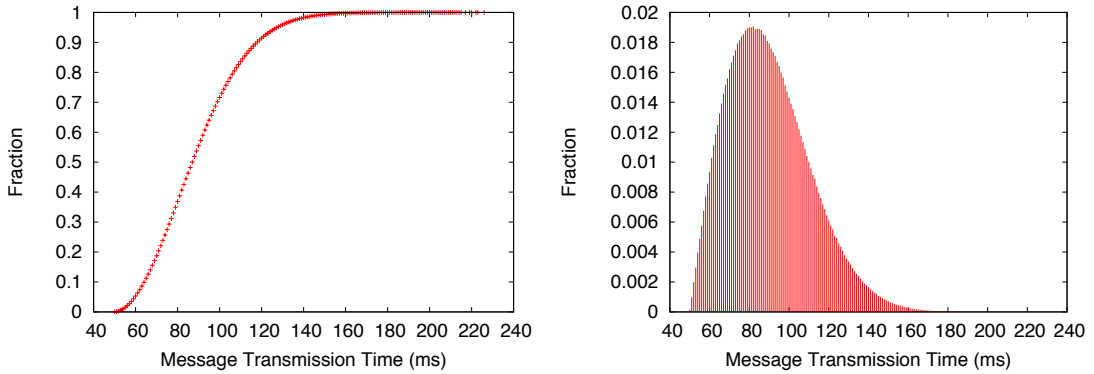
works. Rounds are no longer executed in lock-step and individual nodes can transmit at different times. An additional timeout parameter τ is used by FU to set the maximum time that each node will wait for messages from its neighbors, before executing the next iteration of the algorithm (according to the strategy defined in Section 5.3.3). In particular, different timeouts are compared, considering the execution of FU on random topologies ($n = 1000$, $d \approx 3$) with and without message loss. In order to emulate realistic asynchronous settings during the simulation, the transmission time of each sent message was computed according to a predefined probability distribution function (i.e., Weibull), incurring each transmission on some arbitrary delay. The specific parameters of the distribution function were carefully set to approximate the message transmission latencies measured in real environments.

In particular, a rough approximation to the distribution of messages latencies observed in PlanetLab[25; 1] was defined, according to the RTT (Round Trip Time) measurements presented in [123] and [85]. The message transmission times used in the simulation result from the sum of two components: queuing delay, and minimum transmission delay[5]. More precisely, in this case, a Weibull with shape $s = 2$ and scale $r = 45$ was used to generate the queuing delays, and a minimum transmission delay of 50 ms was added. This characterization lead to a message latencies distribution with an average of 89.88 ms, with most of the transmission times below 140 ms, as depicted by Figures 7.30(a) and 7.30(b).

In this settings, timeout values of 25, 50, 100, 125, 150, and 300 ms were considered to evaluate the performance of FU. The results obtained without message loss and with 20% of loss are similar. The best performance in terms of convergence speed is observed for small timeouts (i.e., 25 and 50 ms), and the worst performance for the bigger timeout (i.e., 300 ms), as shown by Figures 7.30(c) and 7.30(e). On the other hand, the use of small timeouts is very inefficient in terms of message load (sending much more messages), and better results are obtained for bigger timeouts, as depicted by Figures 7.30(d) and 7.30(f). The best tradeoff in terms of time and message load is obtained for a timeout of 125 ms, in both scenarios (with and without message loss). Good results are also obtained for timeouts close to 125, namely for 100 and 150 ms.

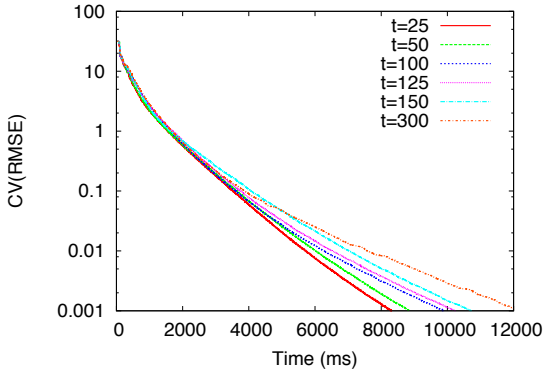
In summary, we observe that the use of different timeouts, according to the messages latencies, influences the performance of FU in asynchronous settings. In particular, the experimental results indicate that better performances are obtained using

[5]Following the method used in [64] to approximate transmission times with a single Weibull.

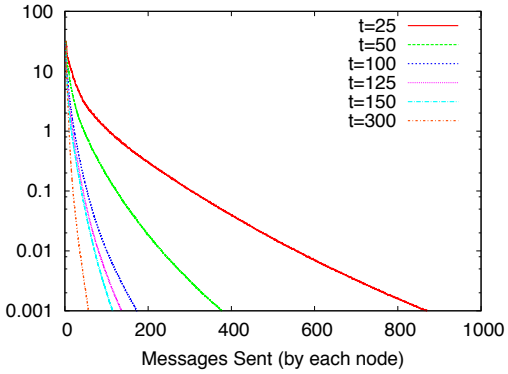


(a) CDF of Message Latencies

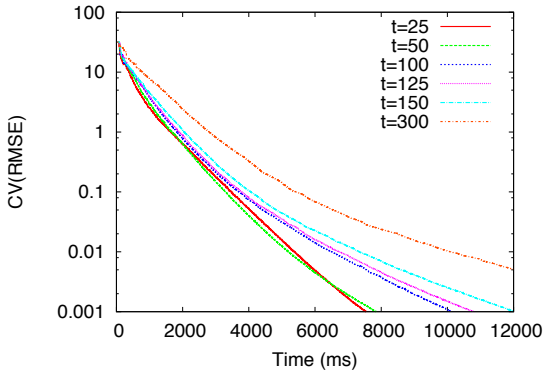
(b) Distribution of Message Latencies



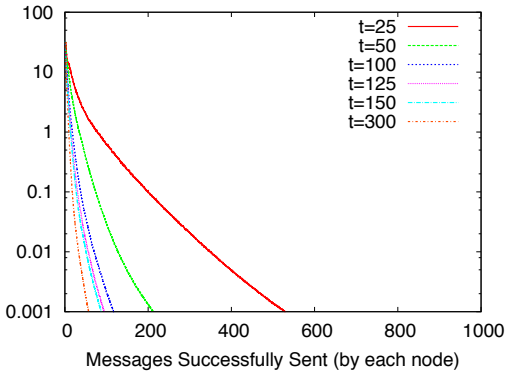
(c) Convergence/Accuracy



(d) Message load



(e) Convergence/Accuracy (20% loss)



(f) Message load (20% loss)

Figure 7.30: Execution of FU in asynchronous setting, using an average message transmission of 89.88 ms – random networks ($n = 1000$, $d \approx 3$).

timeouts above the average transmission times, more precisely timeouts close to the percentiles $95\% - 99\%$ of the underlying message latency distribution. This results appear to be valid independently from the amount of message loss, and were also verified for other message latency distributions and network topologies.

Part V

Peroratio

Achievements and Future Work

Chapter 8

Conclusion

This research work provides an important contribution to the area of distributed computing, more precisely for the distributed computation of aggregation functions (e.g., AVERAGE, COUNT or SUM). In particular, it provides a novel approach, *Flow Updating* (FU), that solves important issues revealed by existing techniques when subject to faults (e.g., message loss) and churn, enabling the effective practical application of a robust data aggregation scheme on realistic settings. FU allows the accurate computation of aggregation functions at all nodes, converging to the correct value over time, independently from the network topology.

This approach distinguishes itself from the existing averaging algorithms by its fault-tolerant capabilities. It solves the mass conservation problem observed on other averaging approaches when subject to message loss, and that affect their correctness leading them to converge to a wrong value. Other approaches require additional mechanisms to detect and restore the lost mass which are not feasible in practice. Even if possible these mechanisms will introduce additional delays and message load, even when no faults occur. In contrast, FU is by design able to support message loss, which only delays it without affecting its convergence to the correct value, and without requiring any additional mechanism. This is achieved by keeping the input values unchanged (by the algorithm) and performing idempotent flow updates. Moreover, FU is resilient to node crash and able to support churn, self-adapting to network changes and modifications of the initial input values. In order to support the departure/crash of nodes, a simple unreliable failure detector is required which is easy to implement in practice. Unlike other algorithms, FU is able to continuously adapt to dynamic changes without requiring any restart protocol.

The empirical evaluation confirmed the robustness of Flow Updating, and its ability to operate in realistic settings. In particular, the results show that FU is able to support message loss rates of 40%, without affecting its correctness, and it is excepted to tolerate even higher levels of losses. Message loss only reduce the performance of FU, in terms of time, in an amount proportional to its effective occurrence. Moreover, FU showed to be able to seamlessly adapt to abrupt changes of the network membership and track continuous variations of the network (i.e., churn and input value changes) with a good accuracy, even in the presence of considerable amounts of message loss. The experiments to evaluate the use of practical failure detectors, to detect node departure/crash, show that FU performs very well with conservative failure detection strategies that minimize the amount of incorrect suspicions (e.g., timeout based implementations with a large timeout). The simulations on asynchronous networks with message loss showed that FU performs better when executed at a pace slower than the average message transmission time between nodes, more precisely when each node waits a time between each iteration that allows the reception of data from the majority of its neighbors.

FU was also compared against other averaging approaches. The results show that FU outperforms existing approaches, especially in networks with a low average connection degree, both in terms of convergence speed and message load (requiring less messages to reach a common accuracy). The performance gap is particularly noticeable in 2D/mesh networks, where FU is much better than other averaging techniques (even those designed specifically for these settings, like DRG).

Although Flow Updating constitutes the main achievement, other contributions emerged from this research work. Namely, a rigorous definition of the aggregation problem is introduced, categorizing the different types of aggregation functions. The state of the art on distributed data aggregation algorithms is surveyed, and a taxonomy is proposed to classify the existing approaches according to different perspectives (i.e., communication and computation). A study of the dependability issues of the exiting aggregation techniques is provided. Finally, a novel distributed scheme based on FU is proposed to compute the Cumulative Distribution Function (CDF) of an attribute.

To the best of our knowledge, *Flow Updating* is the most robust distributed aggregation algorithm, fault-tolerant and self-adapting to changes (of the network membership and the input values), that is able to compute aggregation functions at all nodes with an high accuracy on dynamic networks.

8.1 Future Work

Despite the good performance and the distinctive robustness revealed by Flow Updating, some improvement points were identified. For instance, it was already found that it is possible to increase the performance of FU by "deactivating" some links between nodes [73]. In a nutshell, the strategy simply consists on the use of the local knowledge overheard in messages received from neighbors to identify shared neighbors between two nodes, and apply an heuristic for one of the nodes to ignore the shared neighbor, removing local communication cycles.

More recently, a new method has exhibited considerable speedups of the algorithm, according to some preliminary experimental results. This improvement is based on the attribution of different weights on the links between nodes, according to their clustering coefficient so that nodes that have less neighbors in common are given more communication capacity among them, in a sense increasing the "conductance" of the algorithm. This strategy has revealed itself to be very promising and will be one of the main subjects of our future research work. Furthermore, we expect this improvement to solve the performance degradation observed by FU in random and attach networks with high average degrees.

Another promising line of research regards the robust distributed computation of the distribution of an attribute, more precisely estimating its CDF, over dynamic settings (i.e., subject to churn and input value changes). An initial approach based on FU was already introduced in Chapter 6. However, a thorough analysis of this technique must be performed, and a few additional enhancements might as well be required. Nevertheless, this new approach can provide an important support to many other distributed applications, providing each node with a rich information about global attributes, which can leverage the development of new schemes (replacing previous ones that could not take advantage of this global knowledge). The exploration of further techniques, on practical application scenarios, that can take advantage of the support provided by FU is also left for future work.

Finally, an additional feature that will be interesting to have is a query engine able to work on the top of FU. Typically, existing query engines require the support of a specific routing topology (e.g., tree) to operate, incurring in the handicaps of such structures. Therefore, attending to the fault-tolerance capabilities of FU and its independence from the network topology, it will also be desirable to perform an aggregation query across any nodes in a robust way and independently from the routing

structure, maintaining the benefits of FU. For that purpose, an improved termination detection strategy must be developed.

Part VI

Appendices

Appendix A

Modeling *Flow Updating* as a Difference Equation

Here, *Flow Updating* (FU) is characterized as a multi-dimensional, linear dynamical system, represented by a difference equation of the form:

$$x_{t+1} = Ax_t + B, \qquad t = 0, 1, 2, 3, \ldots \tag{A.1}$$

where x_t is an n-dimensional vector of state variables at time t, $x_t \in \mathbb{R}$, A is an $n \times n$ matrix of constant (time-invariant) coefficients $a_{i,j}$, $a_{i,j} \in \mathbb{R}, i, j = 1, 2, \ldots, n$, and B is an n-dimensional time-independent vector with elements b_i, $b_i \in \mathbb{R}, i, j = 1, 2, \ldots, n$ [55; 44].

A discrete time execution of the algorithm is considered, according to the assumptions defined in Section 5.2.1 (i.e., fixed network without faults). It is assumed that FU proceeds according to Algorithm 1 (synchronous model), where all nodes concurrently execute the algorithm at each round t. In this case, message exchanges are not abstracted, containing a pair of values, with the flow and estimate to each neighbor, and it is assumed that they are correctly delivered (no loss). In addition, for convenience, it is considered that the state of each node i holds an additional variable e_i to keep the estimate computed at the end of each round. Taking these considerations into account, in the following sections FU is represented as a difference equation that models the system dynamics. More specifically, two types of difference equations are defined, modeling FU in terms of the messages exchanged across the network, and the state evolution at all nodes over time.

A.1 State Model

In this case, it is considered that the local state of each node i is composed by three main components: a variable v_i that holds the initial input value to be averaged (at this point, we consider that this value is constant and does not change during the iteration process); a set F_i that denote the flow values associated to all neighbors (with elements $f_{i,j}$ where $j \in \mathcal{D}_i$)[1]; a variable e_i that keep the estimated average value of a node at the end of each round.

At the end of each round t, the estimate e_i^t of a node i will result from the update of its local flows, computed in a way such as the new estimate is the same as the new average, i.e., $e_i^t = a$. Now, notice that, at a time t node i computes a using the data received from its neighbor, i.e.: all the received estimates e_j^{t-1}, and the estimate e_i resulting from the estimation function using the flows $-f_{j,i}^{t-1}$ received from its neighbors as input. Therefore, one can rewrite the expression to calculate the new average at a node i for time $t+1$, which corresponds to the value e_i^{t+1}:

$$e_i^{t+1} = \frac{v_i}{|\mathcal{D}_i|+1} + \frac{\sum_{k\in\mathcal{D}_i} e_k^t}{|\mathcal{D}_i|+1} + \frac{\sum_{k\in\mathcal{D}_i} f_{k,i}^t}{|\mathcal{D}_i|+1} \tag{A.2}$$

This equation characterizes the evolution of the estimated values at any nodes i over time t. Now, let's consider the expression to calculate the flow value of a specific neighbor j at the end of each round, which can also be rewritten as (according to the previous considerations):

$$f_{i,j}^{t+1} = -f_{j,i}^t + (e_i^{t+1} - e_j^t) \tag{A.3}$$

Now, substituting e_i^{t+1} in (A.3) by the expression in (A.2), we obtain the equation:

$$\begin{aligned} f_{i,j}^{t+1} &= -f_{j,i}^t + \left(\frac{v_i}{|\mathcal{D}_i|+1} + \frac{\sum_{k\in\mathcal{D}_i} e_k^t}{|\mathcal{D}_i|+1} + \frac{\sum_{k\in\mathcal{D}_i} f_{k,i}^t}{|\mathcal{D}_i|+1} - e_j^t \right) \\ &= \frac{v_i}{|\mathcal{D}_i|+1} + \frac{\sum_{k\in\mathcal{D}_i} e_k^t}{|\mathcal{D}_i|+1} - e_j^t + \frac{\sum_{k\in\mathcal{D}_i} f_{k,i}^t}{|\mathcal{D}_i|+1} - f_{j,i}^t \end{aligned}$$

[1]Note that, for a matter of simplicity and consistency of the variables notation, $f_{i,j}$ is used instead of $F_i(j)$.

$$
\begin{aligned}
&= \frac{v_i}{|\mathcal{D}_i|+1} + \left(\frac{\sum\limits_{k \in \mathcal{D}_i} e_k^t}{|\mathcal{D}_i|+1} - \frac{|\mathcal{D}_i|+1}{|\mathcal{D}_i|+1} e_j^t \right) + \left(\frac{\sum\limits_{k \in \mathcal{D}_i} f_{k,i}^t}{|\mathcal{D}_i|+1} - \frac{|\mathcal{D}_i|+1}{|\mathcal{D}_i|+1} f_{j,i}^t \right) \\
&= \frac{v_i}{|\mathcal{D}_i|+1} + \left(\frac{\sum\limits_{k \in \mathcal{D}_i \wedge k \neq j} e_k^t}{|\mathcal{D}_i|+1} - \frac{|\mathcal{D}_i|}{|\mathcal{D}_i|+1} e_j^t \right) + \left(\frac{\sum\limits_{k \in \mathcal{D}_i \wedge k \neq j} f_{k,i}^t}{|\mathcal{D}_i|+1} - \frac{|\mathcal{D}_i|}{|\mathcal{D}_i|+1} f_{j,i}^t \right)
\end{aligned}
\tag{A.4}
$$

The obtained expression characterizes the evolution of the flow value of a neighbor j at a node i. The conjugation of both equations (A.2) and (A.4) define a coupled system of p-dimensional (where $p = n+n^2$, being n the number of nodes in the network), first order, non-homogeneous difference equations $s^{t+1} = As^t + B$, that represent the state evolution of the algorithm over time t, with:

$$
A = \begin{bmatrix}
0 & a_{e_1,e_2} & \dots & a_{e_1,e_n} & 0 & \dots & 0 & a_{e_1,f_{2,1}} & \dots & a_{e_1,f_{2,n}} & \dots & a_{e_1,f_{n,1}} & \dots & 0 \\
a_{e_2,e_1} & 0 & \dots & a_{e_2,e_n} & 0 & \dots & a_{e_2,f_{1,n}} & 0 & \dots & 0 & \dots & a_{e_2,f_{n,1}} & \dots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & & \vdots & \ddots & \vdots & & \vdots \\
a_{e_n,e_1} & a_{e_n,e_2} & \dots & 0 & 0 & \dots & a_{e_n,f_{1,n}} & a_{e_n,f_{2,1}} & \dots & a_{e_n,f_{2,n}} & \dots & 0 & \dots & 0 \\
0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\
\vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\
0 & a_{f_{1,n},e_2} & \dots & a_{f_{1,n},e_n} & 0 & \dots & 0 & a_{f_{1,n},f_{2,1}} & \dots & a_{f_{1,n},f_{2,n}} & \dots & a_{f_{1,n},f_{n,1}} & \dots & 0 \\
a_{f_{2,1},e_1} & 0 & \dots & a_{f_{2,1},e_n} & 0 & \dots & a_{f_{2,1},f_{1,n}} & 0 & \dots & 0 & \dots & a_{f_{2,1},f_{n,1}} & \dots & 0 \\
\vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots & & \vdots & & \vdots \\
a_{f_{2,n},e_1} & 0 & \dots & a_{f_{2,n},e_n} & 0 & \dots & a_{f_{2,n},f_{1,n}} & 0 & \dots & 0 & \dots & a_{f_{2,n},f_{n,1}} & \dots & 0 \\
\vdots & \vdots & \ddots & \vdots & \vdots & & \vdots & \vdots & & \vdots & \ddots & \vdots & & \vdots \\
a_{f_{n,1},e_1} & a_{f_{n,1},e_2} & \dots & 0 & 0 & \dots & a_{f_{n,1},f_{1,n}} & a_{f_{n,1},f_{2,1}} & \dots & a_{f_{n,1},f_{2,n}} & \dots & 0 & \dots & 0 \\
\vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots & & \vdots & \ddots & \vdots \\
0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0
\end{bmatrix}
$$

$$\text{where} \quad a_{r,c} = \begin{cases} \frac{1}{|\mathcal{D}_i|+1} & \text{if } r = e_i \wedge c = e_j \text{ and } j \in \mathcal{D}_i \\ \frac{1}{|\mathcal{D}_i|+1} & \text{if } r = e_i \wedge c = f_{j,i} \text{ and } j \in \mathcal{D}_i \\ -\frac{|\mathcal{D}_i|}{|\mathcal{D}_i|+1} & \text{if } r = f_{i,j} \wedge c = e_j \text{ and } j \in \mathcal{D}_i \\ \frac{1}{|\mathcal{D}_i|+1} & \text{if } r = f_{i,j} \wedge c = e_k \wedge j \neq k \text{ and } j,k \in \mathcal{D}_i \\ -\frac{|\mathcal{D}_i|}{|\mathcal{D}_i|+1} & \text{if } r = f_{i,j} \wedge c = f_{j,i} \text{ and } j \in \mathcal{D}_i \\ \frac{1}{|\mathcal{D}_i|+1} & \text{if } r = f_{i,j} \wedge c = f_{k,i} \wedge j \neq k \text{ and } j,k \in \mathcal{D}_i \\ 0 & \text{otherwise} \end{cases}, \tag{A.5}$$

$$B = \begin{bmatrix} \frac{v_1}{|\mathcal{D}_1|+1} \\ \frac{v_2}{|\mathcal{D}_2|+1} \\ \vdots \\ \frac{v_n}{|\mathcal{D}_n|+1} \\ \\ \frac{v_1}{|\mathcal{D}_1|+1} \\ \vdots \\ \frac{v_1}{|\mathcal{D}_1|+1} \\ \frac{v_2}{|\mathcal{D}_2|+1} \\ \vdots \\ \frac{v_2}{|\mathcal{D}_2|+1} \\ \vdots \\ \frac{v_n}{|\mathcal{D}_n|+1} \\ \vdots \\ \frac{v_n}{|\mathcal{D}_n|+1} \end{bmatrix}, \tag{A.6}$$

$$\text{and} \quad s^0 = \begin{bmatrix} e_1^0 \\ e_2^0 \\ \vdots \\ e_n^0 \\ \\ f_{1,1}^0 \\ \vdots \\ f_{1,n}^0 \\ f_{2,1}^0 \\ \vdots \\ f_{2,n}^0 \\ \vdots \\ f_{n,1}^0 \\ \vdots \\ f_{n,n}^0 \end{bmatrix} \tag{A.7}$$

A.2 Message Model

In this section, a linear system of first-order difference equations is defined to model the messages exchanges of FU. In particular, considering the flow $f_{i,j}^t$ and estimate e_i^t to be sent from i to j at time $t+1$, it is assumed that the message sent is the sum of these values instead of a pair, i.e.: $m_{i,j}^t = f_{i,j}^t + e_i^t$ (instead of $m_{i,j}^t = (f_{i,j}^t, e_i^t)$).

Then, following the reasoning performed in the previous section to obtain Equation A.2, one can rewrite the expression to compute the estimate of node i at time $t+1$

in terms of the messages sent by its neighbors:

$$\begin{aligned} e_i^{t+1} &= \frac{v_i + \sum_{k \in \mathcal{D}_i} (f_{k,i}^t + e_k^t)}{|\mathcal{D}_i| + 1} \\ &= \frac{v_i + \sum_{k \in \mathcal{D}_i} m_{k,i}^t}{|\mathcal{D}_i| + 1} \end{aligned} \tag{A.8}$$

Now, let's consider Equation A.3, which is used to calculate the flow value stored at node i for each neighbor j, according to its newly computed estimate (i.e., average). In this case, considering that $m_{i,j}^t = f_{i,j}^t + e_i^t$, we get:

$$\begin{aligned} & f_{i,j}^{t+1} = -f_{j,i}^t + (e_i^{t+1} - e_j^t) \\ \Leftrightarrow \quad & f_{i,j}^{t+1} = e_i^{t+1} - e_j^t - f_{j,i}^t \\ \Leftrightarrow \quad & f_{i,j}^{t+1} = e_i^{t+1} - m_{j,i}^t \\ \Leftrightarrow \quad & e_i^{t+1} + f_{i,j}^{t+1} = e_i^{t+1} - m_{j,i}^t + e_i^{t+1} \\ \Leftrightarrow \quad & e_i^{t+1} + f_{i,j}^{t+1} = 2e_i^{t+1} - m_{j,i}^t \\ \Leftrightarrow \quad & m_{i,j}^{t+1} = 2e_i^{t+1} - m_{j,i}^t \end{aligned} \tag{A.9}$$

Then, attending to the fact that the new estimate e_i^{t+1} computed by node i will be sent to all its neighbors, replacing e_i^{t+1} in Equation A.9 by the expression from Equation A.8, we obtain:

$$\begin{aligned} m_{i,j}^{t+1} &= 2 \left(\frac{v_i + \sum_{k \in \mathcal{D}_i} m_{k,i}^t}{|\mathcal{D}_i| + 1} \right) - m_{j,i}^t \\ &= \frac{2}{|\mathcal{D}_i| + 1} \left(v_i + \sum_{k \in \mathcal{D}_i} m_{k,i}^t \right) - m_{j,i}^t \\ &= \frac{2}{|\mathcal{D}_i| + 1} v_i + \frac{2}{|\mathcal{D}_i| + 1} \left(\sum_{k \in \mathcal{D}_i} m_{k,i}^t \right) - \frac{|\mathcal{D}_i| + 1}{|\mathcal{D}_i| + 1} m_{j,i}^t \\ &= \frac{2}{|\mathcal{D}_i| + 1} v_i + \frac{2}{|\mathcal{D}_i| + 1} \left(\sum_{k \in \mathcal{D}_i \wedge k \neq j} m_{k,i}^t \right) + \frac{1 - |\mathcal{D}_i|}{|\mathcal{D}_i| + 1} m_{j,i}^t \end{aligned} \tag{A.10}$$

This Equation corresponds to a coupled system of q-dimensional (where $q = n^2$, be-

ing n the number of nodes in the network), first order, non-homogeneous difference equations $m^{t+1} = Am^t + B$, that represent the message exchanges of the algorithm over time t, with:

$$A = \begin{bmatrix} 0 & \dots & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & \dots & 0 & a_{m_{1,n},m_{2,1}} & \dots & a_{m_{1,n},m_{2,n}} & \dots & a_{m_{1,n},m_{n,1}} & \dots & 0 \\ 0 & \dots & a_{m_{2,1},m_{1,n}} & 0 & \dots & 0 & \dots & a_{m_{2,1},m_{n,1}} & \dots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & a_{m_{2,n},m_{1,n}} & 0 & \dots & 0 & \dots & a_{m_{2,n},m_{n,1}} & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & a_{m_{n,1},m_{1,n}} & a_{m_{n,1},m_{2,1}} & \dots & a_{m_{n,1},m_{2,n}} & \dots & 0 & \dots & 0 \\ \vdots & & \vdots & \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & 0 & \dots & 0 & \dots & 0 & \dots & 0 \end{bmatrix}$$

$$\text{where} \quad a_{r,c} = \begin{cases} \frac{1-|\mathcal{D}_i|}{|\mathcal{D}_i|+1} & \text{if } r = m_{i,j} \wedge c = m_{j,i} \text{ and } j \in \mathcal{D}_i \\ \frac{2}{|\mathcal{D}_i|+1} & \text{if } r = m_{i,j} \wedge c = m_{k,i} \wedge j \neq k \text{ and } j,k \in \mathcal{D}_i \\ 0 & \text{otherwise} \end{cases}, \tag{A.11}$$

$$B = \begin{bmatrix} \frac{2}{|\mathcal{D}_1|+1} v_1 \\ \vdots \\ \frac{2}{|\mathcal{D}_1|+1} v_1 \\ \frac{2}{|\mathcal{D}_2|+1} v_2 \\ \vdots \\ \frac{2}{|\mathcal{D}_2|+1} v_2 \\ \vdots \\ \frac{2}{|\mathcal{D}_n|+1} v_n \\ \vdots \\ \frac{2}{|\mathcal{D}_n|+1} v_n \end{bmatrix}, \tag{A.12}$$

$$\text{and} \quad m^0 = \begin{bmatrix} m^0_{1,1} \\ \vdots \\ m^0_{1,n} \\ m^0_{2,1} \\ \vdots \\ m^0_{2,n} \\ \vdots \\ m^0_{n,1} \\ \vdots \\ m^0_{n,n} \end{bmatrix} \tag{A.13}$$

A.3 Example

In this section, we show the resulting system of difference equations (both for state and messages) in two concrete and simple examples of the execution of FU. The considered scenarios are similar, and only differ by a small variation of the communication routing topology. More specifically, compared to the first scenario, the topology of the second scenario has an additional link between two nodes. This extra link creates a cycle

and transform the acyclic communication graph of Scenario 1 (see figure A.1) into a unicyclic communication graph in Senario 2 (see figure A.2). In more detail, each considered network is composed of four nodes, with initial values $v_1 = 1$, $v_2 = 2$, $v_3 = 3$, and $v_4 = 6$, which correspond to a global average $\bar{a}$ equal to 3 (i.e., $\bar{a} = (1+2+3+6)/4$). It is assumed that all remaining state variables (flows and estimate) start with a value equal to zero.

A.3.1 Scenario 1 (tree network)

This scenario considers the execution of FU on a network with the topology depicted by Figure A.1.

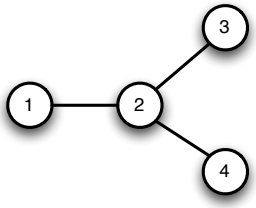


Figure A.1: Tree network topology

The system of difference equations, that represent the state evolution over time in this particular example, is defined by the following matrices:

$$A = \begin{bmatrix}
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{3}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 & -\frac{3}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & 0 & -\frac{3}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{3}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\
\frac{1}{4} & 0 & \frac{1}{4} & -\frac{3}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & -\frac{3}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}, \tag{A.14}$$

$$B = \begin{bmatrix} \frac{1}{2} \\ \frac{2}{4} \\ \frac{3}{2} \\ \frac{6}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{2}{4} \\ \frac{2}{4} \\ \frac{2}{4} \\ \frac{2}{4} \\ \frac{3}{2} \\ \frac{3}{2} \\ \frac{3}{2} \\ \frac{3}{2} \\ \frac{6}{2} \\ \frac{6}{2} \\ \frac{6}{2} \\ \frac{6}{2} \end{bmatrix}, \tag{A.15}$$

$$s^0 = \begin{bmatrix} e_1^0 \\ e_2^0 \\ e_3^0 \\ e_4^0 \\ f_{1,1}^0 \\ f_{1,2}^0 \\ f_{1,3}^0 \\ f_{1,4}^0 \\ f_{2,1}^0 \\ f_{2,2}^0 \\ f_{2,3}^0 \\ f_{2,4}^0 \\ f_{3,1}^0 \\ f_{3,2}^0 \\ f_{3,3}^0 \\ f_{3,4}^0 \\ f_{4,1}^0 \\ f_{4,2}^0 \\ f_{4,3}^0 \\ f_{4,4}^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{A.16}$$

The components of the system of difference equations that represent the message exchanges of FU in this example are:

$$A = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}, \tag{A.17}$$

$$B = \begin{bmatrix} \frac{2}{2} \\ \frac{2}{2} \\ \frac{2}{2} \\ \frac{2}{2} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{6}{2} \\ \frac{6}{2} \\ \frac{6}{2} \\ \frac{6}{2} \\ \frac{12}{2} \\ \frac{12}{2} \\ \frac{12}{2} \\ \frac{12}{2} \end{bmatrix}, \tag{A.18}$$

$$m^0 = \begin{bmatrix} m_{1,1}^0 \\ m_{1,2}^0 \\ m_{1,3}^0 \\ m_{1,4}^0 \\ m_{2,1}^0 \\ m_{2,2}^0 \\ m_{2,3}^0 \\ m_{2,4}^0 \\ m_{3,1}^0 \\ m_{3,2}^0 \\ m_{3,3}^0 \\ m_{3,4}^0 \\ m_{4,1}^0 \\ m_{4,2}^0 \\ m_{4,3}^0 \\ m_{4,4}^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{A.19}$$

Recall that $m_{i,j}^t = f_{i,j}^t + e_i^t$.

A.3.2 Scenario 2 (multi-path network)

In this scenario, FU is executed on a multi-path network with the topology depicted by Figure A.2.

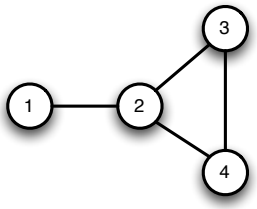


Figure A.2: Multi-path network topology

The system of difference equations, that represent the state evolution of FU over time in this particular example, is defined by the following matrices:

$$A = \begin{bmatrix}
0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\
0 & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 \\
0 & \frac{1}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{3}{4} & 0 & \frac{1}{4} & \frac{1}{4} & 0 & -\frac{3}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & 0 & -\frac{3}{4} & \frac{1}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{3}{4} & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 \\
\frac{1}{4} & 0 & \frac{1}{4} & -\frac{3}{4} & 0 & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & 0 & 0 & 0 & -\frac{3}{4} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{2}{3} & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & -\frac{2}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{3} & 0 & -\frac{2}{3} & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{3} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -\frac{2}{3} & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{2}{3} & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{3} & -\frac{2}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 & 0 & 0 & -\frac{2}{3} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}, \tag{A.20}$$

$$B = \begin{bmatrix} \frac{1}{2} \\ \frac{2}{4} \\ \frac{3}{3} \\ \frac{6}{3} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{1}{2} \\ \frac{2}{4} \\ \frac{2}{4} \\ \frac{2}{4} \\ \frac{2}{4} \\ \frac{3}{3} \\ \frac{3}{3} \\ \frac{3}{3} \\ \frac{3}{3} \\ \frac{3}{3} \\ \frac{3}{3} \\ \frac{6}{3} \\ \frac{6}{3} \\ \frac{6}{3} \\ \frac{6}{3} \end{bmatrix}, \qquad \text{(A.21)} \qquad s^0 = \begin{bmatrix} e_1^0 \\ e_2^0 \\ e_3^0 \\ e_4^0 \\ f_{1,1}^0 \\ f_{1,2}^0 \\ f_{1,3}^0 \\ f_{1,4}^0 \\ f_{2,1}^0 \\ f_{2,2}^0 \\ f_{2,3}^0 \\ f_{2,4}^0 \\ f_{3,1}^0 \\ f_{3,2}^0 \\ f_{3,3}^0 \\ f_{3,4}^0 \\ f_{4,1}^0 \\ f_{4,2}^0 \\ f_{4,3}^0 \\ f_{4,4}^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{(A.22)}$$

For this specific scenario, the components of the system of difference equations that represent the message exchanges of FU are:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -\frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{2}{3} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{3} & 0 & 0 & 0 & \frac{2}{3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{2}{3} & 0 & 0 & 0 & -\frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \qquad \text{(A.23)}$$

$$B = \begin{bmatrix} \frac{2}{2} \\ \frac{2}{2} \\ \frac{2}{2} \\ \frac{2}{2} \\ \frac{2}{2} \\ \frac{2}{2} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{4}{4} \\ \frac{6}{3} \\ \frac{6}{3} \\ \frac{6}{3} \\ \frac{6}{3} \\ \frac{12}{3} \\ \frac{12}{3} \\ \frac{12}{3} \\ \frac{12}{3} \end{bmatrix}, \qquad \text{(A.24)} \qquad m^0 = \begin{bmatrix} m_{1,1}^0 \\ m_{1,2}^0 \\ m_{1,3}^0 \\ m_{1,4}^0 \\ m_{2,1}^0 \\ m_{2,2}^0 \\ m_{2,3}^0 \\ m_{2,4}^0 \\ m_{3,1}^0 \\ m_{3,2}^0 \\ m_{3,3}^0 \\ m_{3,4}^0 \\ m_{4,1}^0 \\ m_{4,2}^0 \\ m_{4,3}^0 \\ m_{4,4}^0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{(A.25)}$$

Once again, recall that $m_{i,j}^t = f_{i,j}^t + e_i^t$.

A.4 Problem

We want to define and prove the stability properties (convergence) of *Flow Updating*, independently from the considered network (size and topology), in order to prove the correctness of the algorithm. In other words, we seek to demonstrate that the value estimated by each node converges over time to the correct global average, considering the defined model.

So far, the Empirical results obtained from several simulations suggest the correctness of the algorithm, independently from the considered network. Next, Some additional properties and observations will be pointed out, which might help in further studies of the stability properties of the algorithm.

A.4.1 Additional information

A.4.1.1 Exploration

In order to prove the stability properties of FU, Corollary A.4.1.1 was found, which generalize the analysis of globally stable systems, based one the eigenvalues of the matrix A, as long as $[I - A]$ is non-singular, meaning that the system has a unique steady-state equilibrium $\bar{x} = [I - A]^{-1}B$.

Corollary A.4.1. (pag. 86 of [55]): Consider the system $x_{t+1} = Ax_t + B$, where $x_t \in \mathbb{R}^n$, and suppose that $\mid I - A \mid \neq 0$. Then, the steady-state equilibrium $\bar{x} = [I - A]^{-1}B$ is globally (asymptotically) stable if and only if the modulus of each eigenvalue of the matrix A is smaller than 1.

Then, in order to verify the compliance of the defined systems of difference equations with Corollary A.4.1.1, a specific analysis of the examples described in Section A.3 was performed:

Scenario 1 - In both models (state and message), the determinant of $[I - A]$ is equal to 1 ($\mid I - A \mid \neq 0$), meaning that $[I - A]$ is non-singular, and both systems have a unique steady-state equilibrium $\bar{x} = [I - A]^{-1}B$. For the state model, the unique steady-state equilibrium $\bar{x}_s$ is the vector $\{3, 3, 3, 3, 1/2, -2, 1/2, 1/2, 2, 1/2, 0, -3, 3/2, 0, 3/2, 3/2, 3, 3, 3, 3\}$, representing the state to which all nodes

will converge. From the resulting vector $\bar{x}_s$, we can easily observe that the resulting estimate is the same for all nodes, and it corresponds to the global average ($\bar{a} = 3$) as expected. For the message model, the unique steady-state equilibrium $\bar{x}_m$ is the vector $\{1, 1, 1, 1, 5, 1, 3, 0, 3, 3, 3, 3, 6, 6, 6, 6\}$, indicating that at some point in time the nodes will exchange the same messages, i.e. the system will converge. To check the resulting estimate at each node, we have to use an auxiliary functions to calculate it from the exchanged message values. Therefore, from Equation (A.9), we can derive a function to determine the estimate $\bar{e}_i$ at the steady-state equilibrium of each node i, based on the exchanged messages between two nodes at that point ($\bar{m}_{i,j}$ and $\bar{m}_{j,i}$)[2]: $\bar{e}_i = (\bar{m}_{i,j} + \bar{m}_{j,i})/2$. Using this function, we can verify that the value estimated by all nodes is the same, and it is equal to the global average (e.g., for node 2: $3 = (5+1)/2 = (3+3)/2 = (0+6)/2$). Finally, in both models all the eigenvalues of matrix A are equal to zero (the absolute value of each eigenvalue is smaller than 1). Therefore, from Corollary A.4.1.1, we can conclude that the system is globally (asymptotically) stable (according to both models).

Scenario 2 - In this scenario, the determinant of $[I - A]$ is equal to 0 for both models (state and message). This result invalidates the application of Corollary A.4.1.1, since one of its conditions $\mid I - A \mid \neq 0$ is not met. The steady-state equilibrium cannot be calculated by $\bar{x} = [I - A]^{-1}B$, because $[I - A]$ is singular and cannot be inverted. Moreover, several simulations of both systems, using different initial values for vectors s^0 and m^0, have suggested the existence of different equilibrium points, all estimating the correct value. For instance, initializing all the vectors elements with the value zero (i.e., all flows and estimates start with the value zero), we obtain a steady-state equilibrium $\bar{x}_s$ equal to $\{3, 3, 3, 3, 1/2, -2, 1/2, 1/2, 2, 1/2, -1, -2, 1, 1, 1, -1, 2, 2, 1, 2\}$ and $\bar{x}_m$ equal to $\{1, 1, 1, 1, 5, 1, 2, 1, 2, 4, 2, 2, 4, 5, 4, 4\}$, for the state and message model respectively. Then, introducing a small change in the initial vectors, simply by considering that $f^0_{2,3} = v_2$ (i.e., $f^0_{2,3}$ and $m^0_{2,3}$ are equal to 2, and all remaining elements keep their value equal to 0), we obtain different steady-state equilibriums $\bar{x}_s$ and $\bar{x}_m$, with the respective values $\{3, 3, 3, 3, 1/2, -2, 1/2, 1/2, 2, 1/2, -2/3, -7/3, 1, 2/3, 1, -2/3, 2, 7/3, 2/3, 2\}$ and $\{1, 1, 1, 1, 5, 1, 7/3, 2/3, 2, 11/3, 2, 7/3,$

[2]Supposing that, if the steady-state equilibrium is reached at time t, the resulting values at time $t+1$ will be kept the same (unless a perturbation is introduced in the system).

$4, 16/3, 11/3, 4\}$. Curiously, if we consider that the initial value of $f^0_{3,2}$ is also equal to v^0_2 (i.e., $f^0_{2,3}$, $f^0_{3,2}$, $m^0_{2,3}$ and $m^0_{3,3}$ are equal to 2), we get the same steady-state equilibriums $\bar{x}_s$ and $\bar{x}_m$ that was obtained when all elements were initialized with zero. Comparing the steady-state vectors resulting from the last situations, we observe that only the elements referring to flows between the nodes 2, 3, 4 (which belong to the graph cycle) exhibit different values. These simulation results suggest that the creation of an "imbalance" between the initial flow values will make the system converge to a different steady-state equilibrium[3]. The correctness of all steady-state equilibriums obtained by simulation was confirmed, simply by verifying the equality $\bar{x} = A.\bar{x} + B$. However, considering the existence of multiple steady-state equilibriums (possibly an infinity of them), will they all correspond to desired states (i.e., with the correct estimate at all nodes)? On the other hand, analyzing the eigenvalues of matrix A, both models have similar eigenvalues (see Section A.4.1.2), all with an absolute value less or equal to one[4], i.e.: 1, $\frac{1}{3}(-1 + i\sqrt{2})$, $\frac{1}{3}(-1 - i\sqrt{2})$, $-\frac{1}{3}$, and 0. But, what can we conclude exactly from the fact that all eigenvalues are smaller than 1, except one which is equal to 1? Will the system converge?

In conclusion, the analysis performed for the specific examples of Section A.3 has revealed different behaviors of the system of equations characterizing FU, which invalidates the use of some tools (e.g., Corollary A.4.1.1) in some scenarios, appearing to be dependent from the network topology (graph structure). The empirical results obtained by simulation (i.e., iterating the systems of difference equations over time, with different inputs) suggest that the system converge to a steady-state equilibrium (even if it is not unique), where the estimates of all nodes correspond to the correct value (i.e., network average). However, how can we demonstrate this for a generic network topology (any graph structure)? Which mathematical tools (e.g., Theorems) could we use? The answers to these questions are left for future work.

[3]From a practical point of view, it suggest that in a multi-path networks, there are several distinct distributions of flows that yield a correct result. This supports the observation, that even the occurrence of some perturbations ("imbalances" due to message losses and slow links) will lead to a correct result (global network average) with a different flow distribution across the network.

[4]Considering the geometrical representation of a pair of complex eigenvalues, $\mu_j \equiv \alpha_j + \beta_j i$ and $\bar{\mu_j} \equiv \alpha_j - \beta_j i$, in the cartesian space. The modulus r_j of the j^{th} eigenvalue is given by $\sqrt{(\alpha_j^2 + \beta_j^2)}$ (see pag. 79 of [55]).

A.4.1.2 Similarity between models: State *vs* Message

Similar results were found from the previous empirical analysis of the examples of Section A.3, for both systems of difference equations: state and message. In particular, common eigenvalues were found for the matrix A that characterize both models, for the same scenario:

Scenario 1 - The eigenvalues of matrix A are $\{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}$ in the state model, and $\{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0\}$ in the message model;

Scenario 2 - The eigenvalues of matrix A are $\{1, \frac{1}{3}(-1+i\sqrt{2}), \frac{1}{3}(-1-i\sqrt{2}), -\frac{1}{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ in the state model, and $\{1, \frac{1}{3}(-1+i\sqrt{2}), \frac{1}{3}(-1-i\sqrt{2}), -\frac{1}{3}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0\}$ in the message model;

A more meticulous observation of both matrices, allowed us to create a relation between the matrix A of both models of difference equations. In a sense, we can see the matrix A of the message model as a component of the corresponding matrix in the state model. For instance, Let A^s be the matrix A of the system of difference equations that models the state evolution, and A^m the matrix A of the system of difference equations that models the message exchange. One can relate the matrix A^s (defined in (A.5)) to A^m (defined in (A.11)), in the following way:

$$A^s = \left[\begin{array}{ccc|ccc} A^s_{e_1,e_1} & \dots & A^s_{e_1,e_n} & A^s_{e_1,f_{1,1}} & \dots & A^s_{e_1,f_{n,n}} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ A^s_{e_n,e_1} & \dots & A^s_{e_n,e_n} & A^s_{e_n,f_{1,1}} & \dots & A^s_{e_n,f_{n,n}} \\ \hline A^s_{f_{1,1},e_1} & \dots & A^s_{f_{1,1},e_n} & & & \\ \vdots & \ddots & \vdots & & A^{m'} & \\ A^s_{f_{n,n},e_1} & \dots & A^s_{f_{n,n},e_n} & & & \end{array}\right]$$

$$\text{and} \quad A^{m'}_{r,c} = \begin{cases} A^m_{r,c} - \frac{1}{|\mathcal{D}_i|+1} & \text{if } A^m_{r,c} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

In other words, the lower right sub-matrix of A^s is equal to the matrix A^m minus $\frac{1}{|\mathcal{D}_i|+1}$, for all elements different from zero.

A.4.1.3 Properties of matrix A

In this section, we simply list some properties of the matrix A that might be found useful in a further proof of the system stability.

- In the state model, the absolute value of all elements $a_{r,c}$ in the matrix A lies in the interval $[0, 1[$. In more detail, all the elements of A that correspond to linked nodes have a value, such that $-1 < a_{r,c} < 0 \wedge 0 < a_{r,c} < 1$, and $a_{r,c} = 0$ otherwise (i.e., $0 \leq |a_{r,c}| < 1$);
- In the message model, the absolute value of all elements $a_{r,c}$ in the matrix A lies in the interval $[0, 1]$. In more detail, all the elements of A that correspond to linked nodes have a value, such that $-1 < a_{r,c} \leq 0 \wedge 0 < a_{r,c} \leq 1$, and $a_{r,c} = 0$ otherwise (i.e., $0 \leq |a_{r,c}| \leq 1$);
- In both models, the trace of the matrix A is equal to zero (i.e., $tr(A) = 0$), since the value of all elements in the main diagonal is equal to zero;
- Matrix norms of A:
  - In the message model, $|| A ||_{max} \leq 1$;
  - In the state model, $|| A ||_{max} < 1$;

Bibliography

[1] Planetlab. `http://www.planet-lab.org` (Last accessed: Aug. 2011).

[2] Ittai Abraham and Dahlia Malkhi. Probabilistic quorums for dynamic systems. *Distributed Computing*, 18(2):113–124, 2005.

[3] Paulo Sérgio Almeida, Carlos Baquero, Martín Farach-Colton, Paulo Jesus, and Miguel A Mosteiro. Fault-Tolerant Aggregation: Flow Update Meets Mass Distribution. In *15th International Conference On Principles Of Distributed Systems (OPODIS)*, Toulouse, France, December 2011 (Accepted).

[4] S Alouf, E Altman, and P Nain. Optimal on-line estimation of the size of a dynamic multicast group. In *21st Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1109–1118, 2002.

[5] Lorenzo Alvisi, Jeroen Doumen, Rachid Guerraoui, Boris Koldehofe, Harry Li, Robbert Renesse, and Gilles Tredan. How robust are gossip-based communication protocols? *ACM SIGOPS Operating Systems Review*, 41(5):14–18, 2007.

[6] Hani Alzaid, Ernest Foo, and Juan Nieto. Secure data aggregation in wireless sensor network: a survey. In *6th Australasian conference on Information security (AISC)*, pages 93–105, 2008.

[7] Sidra Aslam, Farrah Farooq, and Shahzad Sarwar. Power consumption in wireless sensor networks. In *Proceedings of the 7th International Conference on Frontiers of Information Technology (FIT)*, pages 14:1–14:9, Abbottabad, Pakistan, 2009. Punjab University College of Information Technology (PUCIT), University of the Punjab, Anarkali, Lahore, Pakistan.

[8] C Baquero, P Almeida, and R Menezes. Fast Estimation of Aggregates in Unstructured Networks. In *5th International Conference on Autonomic and Autonomous Systems (ICAS)*, pages 88–93, Valencia, Spain, 2009.

[9] Carlos Baquero, Paulo Sérgio Almeida, Raquel Menezes, and Paulo Jesus. Extrema propagation: Fast distributed estimation of sums and network sizes. *IEEE Transactions on Parallel and Distributed Systems*, 99(PrePrints), 2011.

[10] Carlos Baquero, Paulo Sérgio Almeida, Raquel Menezes, and Paulo Jesus. Extrema Propagation: Fast Distributed Estimation of Sums and Network Sizes. *IEEE Transactions on Parallel and Distributed Systems*, (PrePrints), 2011.

[11] Ziv Bar-Yossef, Roy Friedman, and Gabriel Kliot. RaWMS - Random Walk Based Lightweight Membership Service for Wireless Ad Hoc Networks. *ACM Transactions on Computer Systems*, 26(2), 2008.

[12] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 286(5439):509–512, 1999.

[13] M Bawa, H Garcia-Molina, A Gionis, and R Motwani. Estimating aggregates on a peer-to-peer network. Technical report, Stanford University, Computer Science Department, 2003.

[14] Mayank Bawa, Aristides Gionis, Hector Garcia-Molina, and Rajeev Motwani. The price of validity in dynamic networks. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 515–526, 2004.

[15] Yitzhak Birk, Idit Keidar, Liran Liss, and Assaf Schuster. Efficient Dynamic Aggregation. In *20th International Symposium on DIStributed Computing (DISC), LNCS*, pages 90–104, 2006.

[16] Yitzhak Birk, Idit Keidar, Liran Liss, Assaf Schuster, and Ran Wolff. Veracity Radius: Capturing the Locality of Distributed Computations. In *25th annual ACM symposium on Principles of Distributed Computing (PODC)*, 2006.

[17] Miguel Borges, Carlos Baquero, Paulo Jesus, and Paulo Sérgio Almeida. Estimativa Contínua e Tolerante a Faltas de Funções Distribuição Cumulativa em Redes de Larga Escala. In *Simpósio de Informática (INForum)*, 2011.

[18] Jorge C Cardoso, Carlos Baquero, and Paulo Sérgio Almeida. Probabilistic Estimation of Network Size and Diameter. In *4th Latin-American Symposium on Dependable Computing (LADC)*, pages 33–40, 2009.

[19] Tushar Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM*, 43(2):225–267, 1996.

[20] Jen-Yeu Chen and Jianghai Hu. Analysis of Distributed Random Grouping for Aggregate Computation on Wireless Sensor Networks with Randomly Changing Graphs. *IEEE Transactions on Parallel and Distributed Systems*, 19(8):1136–1149, 2008.

[21] Jen-Yeu Chen, G Pandurangan, and Dongyan Xu. Robust Computation of Aggregates in Wireless Sensor Networks: Distributed Randomized Algorithms and Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 17(9):987–1000, 2006.

[22] Siyao Cheng, Jianzhong Li, Qianqian Ren, and Lei Yu. Bernoulli Sampling Based (ε, δ)-Approximate Aggregation in Large-Scale Sensor Networks. In *Proceedings of the 29th IEEE conference on Information communications (INFOCOM)*, pages 1181–1189, 2010.

[23] Laukik Chitnis, Alin Dobra, and Sanjay Ranka. Aggregation Methods for Large-Scale Sensor Networks. *ACM Transactions on Sensor Networks (TOSN)*, 4(2):1–36, 2008.

[24] Laukik Chitnis, Alin Dobra, and Sanjay Ranka. Fault tolerant aggregation in heterogeneous sensor networks. *Journal of Parallel and Distributed Computing*, 69(2):210–219, 2009.

[25] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[26] E Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[27] Edith Cohen and Haim Kaplan. Spatially-decaying aggregation over a network: model and algorithms. In *Proceedings of the ACM SIGMOD international conference on Management of data*, 2004.

[28] Edith Cohen and Haim Kaplan. Summarizing data using bottom-k sketches. In *26th annual ACM symposium on principles of distributed computing (PODC)*, pages 225–234, 2007.

[29] J Considine, F Li, G Kollios, and J Byers. Approximate aggregation techniques for sensor databases. In *20th International Conference on Data Engineering (ICDE)*, pages 449–460, 2004.

[30] M Dam and R Stadler. A Generic Protocol for Network State Aggregation. *Radiovetenskap och Kommunikation (RVK)*, 2005.

[31] A DasGupta. The matching, birthday and the strong birthday problem: a contemporary review. *Journal of Statistical Planning and Inference*, 130(1-2):377–389, 2005.

[32] B. A. Davey and Hilary A. Priestley. *Introduction to lattices and order*. Cambridge University Press, Cambridge, 2nd edition, 2002.

[33] A Deligiannakis, Y Kotidis, and N Roussopoulos. Hierarchical In-Network Data Aggregation with Quality Guarantees. In *International Conference on Extending Database Technology (EDBT)*, pages 658–675, 2004.

[34] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th annual ACM Symposium on Principles of distributed computing (PODC)*, 1987.

[35] Reinhard Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 4th edition edition, 2010.

[36] A Dimakis, A Sarwate, and M Wainwright. Geographic gossip: efficient aggregation for sensor networks. In *5th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 69–76, 2006.

[37] A Dimakis, A Sarwate, and M Wainwright. Geographic Gossip: Efficient Averaging for Sensor Networks. *IEEE Transactions on Signal Processing*, 56(3):1205–1216, 2008.

[38] Monica Dixit and Antonio Casimiro. Adaptare-FD: A Dependability-Oriented Adaptive Failure Detector. In *29th IEEE Symposium on Reliable Distributed Systems*, pages 141–147, 2010.

[39] D Dolev, O Mokryn, and Y Shavitt. On multicast trees: structure and size estimation. In *22nd Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1011–1021, 2003.

[40] D Dolev, O Mokryn, and Y Shavitt. On multicast trees: structure and size estimation. *IEEE/ACM Transactions on Networking*, 14(3):557–567, 2006.

[41] S Dolev, E Schiller, and J Welch. Random Walk for Self-Stabilizing Group Communication in Ad-Hoc Networks. In *21st IEEE Symposium on Reliable Distributed Systems*, pages 70–79, 2002.

[42] S Dolev, E Schiller, and J Welch. Random Walk for Self-Stabilizing Group Communication in Ad Hoc Networks. *IEEE Transactions on Mobile Computing*, 5(7):893–905, 2006.

[43] Marianne Durand and Philippe Flajolet. Loglog Counting of Large Cardinalities (Extended Abstract). In *11th Annual European Symposium on Algorithms (ESA)*, pages 605–617, Budapest, Hungary, 2003.

[44] Saber Elaydi. *An Introduction to Difference Equations*. Undergraduate Texts in Mathematics. Springer, 3rd edition, 2005.

[45] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61, 1960.

[46] P Eugster, R Guerraoui, S Handurukande, P Kouznetsov, and A-M Kermarrec. Lightweight probabilistic broadcast. *ACM Transactions on Computer Systems*, 21(4):341–374, 2003.

[47] Ittay Eyal, Idit Keidar, and Raphael Rom. Distributed Clustering for Robust Aggregation in Large Networks. In *5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.

[48] Ittay Eyal, Idit Keidar, and Raphael Rom. Distributed data classification in sensor networks. In *Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing (PODC)*, pages 151–160, 2010.

[49] Yao-Chung Fan and A Chen. Efficient and robust sensor data aggregation using linear counting sketches. In *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2008.

[50] Yao-Chung Fan and Arbee L Chen. Efficient and Robust Schemes for Sensor Data Aggregation Based on Linear Counting. *IEEE Transactions on Parallel and Distributed Systems*, 21(11):1675–1691, 2010.

[51] E Fasolo, M Rossi, J Widmer, and M Zorzi. In-network aggregation techniques for wireless sensor networks: a survey. *IEEE Wireless Communications*, 14(2):70–87, 2007.

[52] P Flajolet and G Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 31(2):182–209, 1985.

[53] Philippe Flajolet, Eric Fusy, Olivier Gandouet, and Frederic Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *International Conference on Analysis of Algorithms (AofA)*, pages 127–146, 2007.

[54] T Friedman and D Towsley. Multicast session membership size estimation. In *18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 965–972, 1999.

[55] Oded Galor. *Discrete Dynamical Systems*. Springer, 1st edition, 2006.

[56] A Ganesh, A Kermarrec, E Le Merrer, and L Massoulié. Peer counting and sampling in overlay networks based on random walks. *Distributed Computing*, 20(4):267–278, 2007.

[57] A Ganesh, A Kermarrec, and L Massoulie. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, 2003.

[58] Jim Gray. Notes on Data Base Operating Systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating Systems: An Advanced Course*, pages 393–481. Operating Systems, 1978.

[59] Michael Greenwald and Sanjeev Khanna. Power-conserving computation of order-statistics over sensor networks. In *Proceedings of the 23rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS)*, pages 275–285, 2004.

[60] Indranil Gupta, Robbert Van Renesse, and Kenneth P Birman. Scalable fault-tolerant aggregation in large process groups. In *International Conference on Dependable Systems and Networks (DSN)*, pages 433–442, 2001.

[61] M Haridasan and R van Renesse. Gossip-based distribution estimation in peer-to-peer networks. In *International workshop on Peer-To-Peer Systems (IPTPS)*, 2008.

[62] W Heinzelman, A Chandrakasan, and H Balakrishnan. Energy-Efficient Communication Protocol for Wireless Microsensor Networks. In *33rd Annual Hawaii International Conference on System Sciences*, page 10, 2000.

[63] W Heinzelman, A Chandrakasan, and H Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE Transactions on Wireless Communications*, 1(4):660–670, 2002.

[64] J Hernandez and I Phillips. Weibull mixture model to characterise end-to-end Internet delay at coarse time-scales. *IEE Proceedings - Communications*, 153(2):295–304, 2006.

[65] Keren Horowitz and Dahlia Malkhi. Estimating network size from local information. *Information Processing Letters*, 88(5):237–243, 2003.

[66] M Jelasity, W Kowalczyk, and M van Steen. An Approach to Massively Distributed Aggregate Computing on Peer-to-Peer Networks. In *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 200–207, 2004.

[67] M Jelasity and A Montresor. Epidemic-style proactive aggregation in large overlay networks. In *24th International Conference on Distributed Computing Systems*, pages 102–109, 2004.

[68] M Jelasity, A Montresor, and O Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems*, 23(3):219–252, 2005.

[69] Márk Jelasity, Alberto Montresor, Gian Paolo Jesi, and Spyros Voulgaris. The Peersim simulator. `http://peersim.sf.net` (Last accessed: Aug. 2011).

[70] Paulo Jesus, Carlos Baquero, and Paulo Almeida. ID Generation in Mobile Environments. In *Conference on Mobile and Ubiquitous Systems (CSMU)*, pages 159–162. University Of Minho, 2006.

[71] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Dependability in Aggregation by Averaging. In *Simpósio de Informática (INForum)*, 2009.

[72] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Fault-Tolerant Aggregation by Flow Updating. In *9th IFIP International Conference on Distributed Applications and interoperable Systems (DAIS)*, pages 73–86, 2009.

[73] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. Using Less Links to Improve Fault-Tolerant Aggregation. In *[Fast Abstract] 4th Latin-American Symposium on Dependable Computing (LADC)*, 2009.

[74] Paulo Jesus, Carlos Baquero, and Paulo Sergio Almeida. Fault-Tolerant Aggregation for Dynamic Networks. In *29th IEEE Symposium on Reliable Distributed Systems*, pages 37–43, 2010.

[75] Lujun Jia, Guevara Noubir, Rajmohan Rajaraman, and Ravi Sundaram. GIST: Group-Independent Spanning Tree for Data Aggregation in Dense Sensor Networks. In Phillip Gibbons, Tarek Abdelzaher, James Aspnes, and Ramesh Rao, editors, *Distributed Computing in Sensor Systems*, pages 282–304. Springer Berlin / Heidelberg, 2006.

[76] T Jurdziński, Miroslaw Kutylowski, and J Zatopiański. Energy-Efficient Size Approximation of Radio Networks with No Collision Detection. In *8th Annual*

International Conference on Computing and Combinatorics (COCOON), pages 279–289, 2002.

[77] M Frans Kaashoek and David R Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In *2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 98–107. MIT Laboratory for Computer Science, 2003.

[78] J Kabarowski, M Kutyłowski, and W Rutkowski. Adversary Immune Size Approximation of Single-Hop Radio Networks. In *3th International Conference on Theory and Applications of Models of Computation (TAMC)*, pages 148–158, 2006.

[79] S Kashyap, S Deb, K Naidu, R Rastogi, and A Srinivasan. Gossip-Based Aggregate Computation with Low Communication Overhead. In *12th International Telecommunications Network Strategy and Planning Symposium*, pages 1–6, 2006.

[80] D Kempe, A Dobra, and J Gehrke. Gossip-Based Computation of Aggregate Information. In *44th Annual IEEE Symposium on Foundations of Computer Science*, pages 482–491, 2003.

[81] Oliver Kennedy, Christoph Koch, and Al Demers. Dynamic Approaches to In-network Aggregation. In *IEEE 25th International Conference on Data Engineering (ICDE)*, pages 1331–1334, 2009.

[82] Anne-Marie Kermarrec and Maarten Steen. Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review*, 41(5):2–7, 2007.

[83] D Kostoulas, D Psaltoulis, Indranil Gupta, K Birman, and Al Demers. Decentralized Schemes for Size Estimation in Large and Dynamic Groups. In *4th IEEE International Symposium on Network Computing and Applications*, pages 41–48, 2005.

[84] Dionysios Kostoulas, Dimitrios Psaltoulis, Indranil Gupta, Kenneth P Birman, and Alan J Demers. Active and passive techniques for group size estimation in large-scale and dynamic distributed systems. *Elsevier Journal of Systems and Software*, 80(10):1639–1658, 2007.

[85] Jonathan Ledlie, Paul Gardner, and Margo Seltzer. Network Coordinates in the Wild. In *Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, pages 299–311, 2007.

[86] J Li, K Sollins, and D Lim. Implementing Aggregation and Broadcast over Distributed Hash Tables. *ACM SIGCOMM Computer Communication Review*, 35(1):81–92, 2005.

[87] WH Liao, Y Kao, and CM Fan. Data aggregation in wireless sensor networks using ant colony algorithm. *Journal of Network and Computer Applications*, 31(4):387–401, 2008.

[88] Hong Luo, Yonghe Liu, and Sajal K Das. Distributed Algorithm for En Route Aggregation Decision in Wireless Sensor Networks. *IEEE Transactions on Mobile Computing*, 8(1):1–13, 2009.

[89] Hong Luo, J Luo, Y Liu, and S Das. Adaptive Data Fusion for Energy Efficient Routing in Wireless Sensor Networks. *IEEE Transactions on Computers*, 55(10):1286–1299, 2006.

[90] Nancy A Lynch. *Distributed Algorithms*. 1996.

[91] S Madden, R Szewczyk, M Franklin, and D Culler. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. In *4th IEEE Workshop on Mobile Computing Systems and Applications*, pages 49–58, 2002.

[92] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TAG: a Tiny AGgregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI):131–146, 2002.

[93] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the 21st annual symposium on Principles of distributed computing (PODC)*, pages 183–192, 2002.

[94] Sandeep Mane, Sandeep Mopuru, Kriti Mehra, and Jaideep Srivastava. Network Size Estimation In A Peer-to-Peer Network. Technical report, Department of Computer Science, University of Minnesota, 2005.

[95] Amit Manjhi, Suman Nath, and Phillip Gibbons. Tributaries and Deltas: Efficient and Robust Aggregation in Sensor Network Streams. In *ACM SIGMOD International Conference on Management Of Data*, pages 287–298, 2005.

[96] Gurmeet Manku. Routing networks for distributed hash tables. In *Proceedings of the 22nd annual symposium on Principles of Distributed Computing (PODC)*, 2003.

[97] Gurmeet Singh Manku, Mayank Bawa, Prabhakar Raghavan, and Verity Inc. Symphony: Distributed Hashing in a Small World. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2003.

[98] Laurent Massoulié, Erwan Merrer, Anne-Marie Kermarrec, and Ayalvadi Ganesh. Peer Counting and Sampling in Overlay Networks: Random Walk Methods. In *25th annual ACM symposium on Principles of Distributed Computing (PODC)*, 2006.

[99] M Mehyar, D Spanos, J Pongsajapan, S Low, and R Murray. Asynchronous Distributed Averaging on Communication Networks. *IEEE/ACM Transactions on Networking*, 15(3):512–520, 2007.

[100] Erwan Le Merrer, Anne-Marie Kermarrec, and Laurent Massoulie. Peer to peer size estimation in large and dynamic networks: A comparative study. In *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing*, pages 7–17, 2006.

[101] R Misra and C Mandal. Ant-aggregation: ant colony algorithm for optimal data aggregation in wireless sensor networks. In *IFIP International Conference on Wireless and Optical Communications Networks*, page 5, 2006.

[102] Dragoslav Mitrinovic. *Analytic Inequalities*. Springer-Verlag, 1970.

[103] A Montresor, M Jelasity, and O Babaoglu. Robust aggregation protocols for large-scale overlay networks. In *International Conference on Dependable Systems and Networks (DSN)*, pages 19–28, 2004.

[104] D Mosk-Aoyama and D Shah. Computing Separable Functions via Gossip. In *25th annual ACM symposium on Principles of Distributed Computing (PODC)*, pages 113–122, 2006.

[105] D Mosk-Aoyama and D Shah. Fast Distributed Algorithms for Computing Separable Functions. *IEEE Transactions on Information Theory*, 54(7):2997–3007, 2008.

[106] S Motegi, K Yoshihara, and H Horiuchi. DAG based In-Network Aggregation for Sensor Network Monitoring. In *International Symposium on Applications and the Internet (SAINT)*, page 8, 2005.

[107] Eduardo Nakamura, Antonio Loureiro, and Alejandro Frery. Information fusion for wireless sensor networks: Methods, models, and classifications. *ACM Computing Surveys (CSUR*, 39(3):1–55, 2007.

[108] Suman Nath, Phillip Gibbons, Srinivasan Seshan, and Zachary Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, 2004.

[109] N Ntarmos, P Triantafillou, and G Weikum. Counting at Large: Efficient Cardinality Estimation in Internet-Scale Data Networks. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, 2006.

[110] N Ntarmos, P Triantafillou, and G Weikum. Distributed hash sketches: Scalable, efficient, and accurate cardinality estimation for distributed multisets. *ACM Transactions on Computer Systems*, 27(1), 2009.

[111] Boris Pittel. On Spreading a Rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.

[112] Alberto Prieto and Rolf Stadler. Adaptive Distributed Monitoring with Accuracy Objectives. In *SIGCOMM Workshop on Internet Network Management (INM)*, pages 65–70, 2006.

[113] R Rajagopalan and P Varshney. Data-aggregation techniques in sensor networks: a survey. *IEEE Communications Surveys & Tutorials*, 8(4):48–63, 2005.

[114] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 161–172, 2001.

[115] Antony Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg (Middleware)*, pages 329–350, 2001.

[116] J Sacha, J Napper, C Stratan, and G Pierre. Adam2: Reliable Distribution Estimation in Decentralised Environments. In *IEEE 30th International Conference on Distributed Computing Systems (ICDCS)*, pages 697–707, 2010.

[117] Yingpeng Sang, Hong Shen, Y Inoguchi, Yasuo Tan, and Naixue Xiong. Secure Data Aggregation in Wireless Sensor Networks: A Survey. In *7th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 315–320, 2006.

[118] Carl J. Schwarz and George A. F. Seber. Estimating Animal Abundance: Review III. *Statistical Science*, 14(4):427–456, 1999.

[119] A Sharaf, Jonathan Beaver, Alexandros Labrinidis, and K Chrysanthis. Balancing energy efficiency and quality of aggregate data in sensor networks. *International Journal on Very Large Data Bases (VLDB)*, 13(4):384–403, 2004.

[120] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys)*, pages 239–249, 2004.

[121] Ion Stoica, Robert Morris, David Karger, M Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM)*, pages 149–160, 2001.

[122] Apostolos Syropoulos. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing: Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, pages 347–358. Democritus University of Thrace Department of Civil Engineering 671 00 Xanthi GR Greece, 2001.

[123] Li Tang, Yin Chen, Fei Li, Hui Zhang, and Jun Li. Empirical Study on the Evolution of PlanetLab. In *Proceedings of the 6th International Conference on Networking (ICN)*, pages 64–70. IEEE, 2007.

[124] Robbert van Renesse. The Importance of Aggregation. In André Schiper, Alex A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, pages 87–92. Springer-Verlag, 2003.

[125] Robbert van Renesse, Kenneth Birman, and Werner Vogels. Astrolabe: A Robust and Scalable Technology for Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[126] Hongpeng Wang and Neng Luo. An Improved Ant-Based Algorithm for Data Aggregation in Wireless Sensor Networks. In *International Conference on Communications and Mobile Computing (CMC)*, pages 239–243, 2010.

[127] Kyu-Young Whang, Brad Vander-Zanden, and Howard Taylor. A linear-time probabilistic counting algorithm for database applications. *ACM Transactions on Database Systems (TODS)*, 15(2):208–229, 1990.

[128] Fetahi Wuhib, Mads Dam, Rolf Stadler, and Alexander Clemm. Robust Monitoring of Network-wide Aggregates through Gossiping. In *10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 226–235, 2007.

[129] B Zhao, J Kubiatowicz, and A Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, University of California, Berkeley, 2001.

[130] J Zhao, R Govindan, and D Estrin. Computing Aggregates for Monitoring Wireless Sensor Networks. In *1st IEEE International Workshop on Sensor Network Protocols and Applications*, pages 139–148, 2003.