Flow Updating: Fault-Tolerant Aggregation for Dynamic Networks

Paulo Jesus[1,*], Carlos Baquero[1,2,**], Paulo Sérgio Almeida[1,3,**]

HASLab, INESC TEC and Universidade do Minho, Portugal.

Abstract

Data aggregation is a fundamental building block of modern distributed systems. Averaging based approaches, commonly designated gossip-based, are an important class of aggregation algorithms as they allow all nodes to produce a result, converge to any required accuracy, and work independently from the network topology. However, existing approaches exhibit many dependability issues when used in faulty and dynamic environments. This paper describes and evaluates a fault tolerant distributed aggregation technique, Flow Updating, which overcomes the problems in previous averaging approaches and is able to operate on faulty dynamic networks. Experimental results show that this novel approach outperforms previous averaging algorithms; it self-adapts to churn and input value changes without requiring any periodic restart, supporting node crashes and high levels of message loss, and works in asynchronous networks. Realistic concerns have been taken into account in evaluating Flow Updating, like the use of unreliable failure detectors and asynchrony, targeting its application to realistic environments.

1. Introduction

With the advent of multi-hop ad-hoc networks, sensor networks and large-scale overlay networks, there is a demand for tools that can abstract meaningful system properties from given assemblies of nodes. In such settings, aggregation plays an essential role in the design of distributed applications [1], allowing the determination of network-wide properties like network size, total storage capacity, average load, and majorities. Although apparently simple, in practice aggregation has revealed itself to be a non-trivial problem in distributed settings, where no single element holds a global view of the whole system.

In the recent years, several algorithms have addressed the problem with diverse approaches, exhibiting different characteristics in terms of accuracy, time and communication trade-offs. A useful class of aggregation algorithms is based on *averaging* techniques. Such algorithms start from a set of input values spread across the network nodes, and iteratively average their values with neighbors. Eventually all nodes will converge to the same value and can estimate some useful metric.

Averaging techniques allow the derivation of different aggregation functions besides average (like counting and summing), according to the initial combinations of input values. For example, if one node starts with input 1 and all other nodes with input 0, eventually all nodes will end up with the same average $1/n$ and the network size n can be directly estimated by all of them [2].

Distributed data aggregation becomes particularly difficult to achieve when faults are taken into account (i.e., message loss and node crashes), and especially if dynamic settings are considered (nodes arriving/leaving). Few have approached the problem under these settings [3, 4, 5, 6, 7,

*Principal corresponding author
**Corresponding authors
Email addresses: pcoj@di.uminho.pt (Paulo Jesus), cbm@di.uminho.pt (Carlos Baquero), psa@di.uminho.pt (Paulo Sérgio Almeida)
[1]Fax: +351 253 604 471
[2]Tel.: +351 253 604 449
[3]Tel.: +351 253 604 451

8], proving to be hard to efficiently obtain accurate and reliable aggregation results in faulty and dynamic environments.

This paper extends the previous work on *Flow Updating* [9, 10], a novel averaging approach, by presenting asynchronous versions of the algorithm and providing extensive evaluation results considering practical concerns such as: dynamic input value changes, realistic failure detectors and asynchronous execution with message loss. The evaluation shows that: it outperforms classic averaging-based aggregation algorithms; it is fault-tolerant (to both message loss and node crashes); it is able to efficiently support network dynamism (churn); it can be used with realistic failure detectors (and shows how these should be tuned); it can continuously aggregate under changes of input values with no need for a restart; it can be used in asynchronous settings, with variable transmission latency (and shows how timeouts can be chosen for a typical latency distribution, in a practical implementation).

The remainder of this paper is organized as follows. We briefly refer to the related work on aggregation algorithms in Section 2. Section 3 describes *Flow Updating*, a robust distributed aggregation algorithm able to work in dynamic networks. In Section 4, we evaluate the proposed approach. Finally, we make some concluding remarks in Section 5.

2. Related Work

Classic approaches, like TAG [3], perform a tree-based aggregation where partial aggregates are successively computed from child nodes to their parents until the root of the aggregation tree is reached (requiring the existence of a specific routing topology). This kind of aggregation technique is often applied in practice to Wireless Sensor Networks (WSN) [11]. Other tree-based aggregation approaches can be found in [4], [12], and [13]. We should point out that, although being energy-efficient, the reliability of these approaches may be strongly affected by the inherent presence of single-points of failure in the aggregation structure. Moreover, in order to operate on dynamic settings, a tree maintenance protocol is required to handle node arrival/departure, which may lead to temporary disconnection during the parent switching process.

Alternative aggregation algorithms based on the application of probabilistic methods can also be found in the literature. This is the case of Extrema Propagation [14] and COMP [15], which reduce the computation of an aggregation function to the determination of the minimum/maximum of a collection of random numbers. These techniques tend to emphasize speed, being less accurate than averaging approaches.

Specialized probabilistic algorithms can also be used to compute specific aggregation functions, such as COUNT (e.g., to determine the network size). This type of algorithm essentially relies on the results from a sampling process to produce an approximate estimate of the aggregate, using properties of random walks, capture-recapture methods and other statistic tools [16, 5, 17, 6]. These approaches can provide some flexibility in dynamic settings, but are not accurate. The estimation error, present even in fault-free settings, depends on the quality of the collected sample, and the used estimator. Moreover, a sample is made available at a single node, and it can take several rounds to collect one sample. For example, the estimation error can reach 20% in Sample & Collide [16, 5], and a single sampling step takes $\bar{d}T$ (where $\bar{d}$ is the average connection degree and T is a timer value that must be sufficiently large to provide a good sample quality) and must be repeated until l new samples have been observed.

The averaging approach to distributed aggregation is based on an iterative averaging process between small sets of nodes [18, 7, 2, 19, 20]. Eventually, all nodes will converge to the correct value by performing the averaging process across all the network. These approaches are independent from the network routing topology, are often based on a gossip (or epidemic) communication scheme, and are

able to produce an estimate of the resulting aggregate at every network node. Averaging techniques are considered to be robust and accurate (converge over time) when compared to other aggregation techniques, but in practice they exhibit relevant problems that have been overlooked, not supporting message loss nor node crashes (see [21] for more details). Moreover, most existing approaches rely on inefficient strategies to handle network dynamism, like the use of a restart mechanism that looses all progress.

A technique which combines the basic idea from Flow Updating with mass distribution is the MDFU algorithm, presented in [22]. This one keeps a pair of incoming-outgoing flow-like values, but which increase unboundedly. It inherits the convergence properties of the underlying mass distribution, while also being fault-tolerant and allowing input value changes. Another algorithm which keeps a pair of incoming-outgoing values that summarize past messages is the more recent Limosense [23], which adapts the classic Push-Sum [18], inheriting its convergence properties, while being also fault-tolerant and allowing input value changes and network dynamism. Recently, an algorithm named Push-Flow that combines Flow Updating with Push-Sum was described in [24].

A comprehensive survey about distributed data aggregation algorithms is found in [25].

3. Flow Updating

Flow Updating [9, 10] is a recent averaging based aggregation approach, which works for any network topology and tolerates faults. Like existing gossip-based approaches, it averages values iteratively during the aggregation process towards converging to the global network average. But unlike them, it is based on the concept of *flow*, providing unique fault-tolerant characteristics by performing idempotent updates.

The key idea in Flow Updating is to use the *flow* concept from graph theory (which serves as an abstraction for many things like water flow or electric current), and instead of storing in each node the current estimate in a variable, compute it from the input value and the contribution of the flows along edges to the neighbors:

$$e_i = v_i - \sum_{j \in n_i} f_{ij}. \tag{1}$$

This 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. The algorithm aims to enforce and explore the skew symmetry property of the flow along an edge, i.e., $f_{ij} = -f_{ji}$.

The essence of the algorithm is: each node i stores the flow f_{ij} to each neighbor j; node i sends flow f_{ij} to j in a message; a node j receiving f_{ij} updates its own f_{ji} with $-f_{ij}$. Messages simply update flows, 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:

$$\sum_{i \in V} e_i = \sum_{i \in V} (v_i - \sum_{j \in n_i} f_{ij}) = \sum_{i \in V} v_i. \tag{2}$$

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 somewhat 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 $f_{ij} + f_{ji}$ 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 convergence direction towards the correct value.

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, not by instantaneous mass invariance, but by having *mass convergence*.

In order to operate on dynamic networks Flow Updating maintains a dynamic mapping of flows according to the current set of neighbors: removing the entries relative to leaving (or crashing) nodes, and adding entries for newly arrived nodes. The averaging process in each node uses only the current set of neighbors. This straightforward strategy allows Flow Updating to cope with node departure/crash and node arrival, extending its fault tolerance properties.

Node departure, crash or arrival are modeled by a failure detector [26] that gives for each node at each moment the set of neighbors considered to be alive. The interesting thing is that Flow Updating allows the use of practical implementations of failure detectors, that can be incorrect many times (falsely suspecting correct nodes or vice-versa) without compromising correctness.

New nodes are immediately allowed to participate in the averaging process, and leaving or crashed nodes implicitly stop participating in it. The algorithm runs continuously, without requiring restarts in order to adapt to network changes/failures, and simply makes use of the set of neighbors n_i given by the failure detector. No concept of *epoch* is required and the algorithm is always converging towards the average according to the current set of participants, allowing a fast self adaptation to network changes. Another advantage of Flow Updating is that it also allows the input values to be aggregated v_i to change over time (e.g. a temperature). Again, the algorithm will converge to the aggregation of the most recent value at each node without requiring a restart.

3.1. Algorithm – Synchronous Version

The algorithm is now described under the synchronous network model (as in Chapter 2 of [27]). Computation proceeds in synchronous rounds. At each round, 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*. Each node needs only to be able to distinguish its neighbors, not requiring the use of globally unique identifiers.

```
1  inputs:
2      v_i, value to aggregate
3      n_i, set of neighbors given by failure detector
4  state:
5      F_i, flows: initially, F_i = {}
6  message-generation function:
7      msg_i(F_i, j) = (i, F_i(j), est(v_i, F_i))
8  state-transition function:
9      trans_i(F_i, M_i) = F_i'
10     with
11     F = {j ↦ −f | j ∈ n_i ∧ (j, f, _) ∈ M_i} ∪
12         {j ↦ f | j ∈ n_i ∧ (j, _, _) ∉ M_i ∧ (j, f) ∈ F_i}
13     E = {i ↦ est(v_i, F)} ∪
14         {j ↦ e | j ∈ n_i ∧ (j, _, e) ∈ M_i} ∪
15         {j ↦ est(v_i, F_i) | j ∈ n_i ∧ (j, _, _) ∉ M_i}
16     a = (∑{e | (_, e) ∈ E})/|E|
17     F_i' = {j ↦ f + a − E(j) | (j, f) ∈ F}
18 estimation function:
19     est(v, F) = v − ∑{f | (_, f) ∈ F}
```

Algorithm 1: Flow Updating algorithm for dynamic networks in the synchronous network model.

The algorithm, presented in Algorithm 1, makes use of *inputs*, which are values that can change due to external factors, whose current value can be read at any round, but are not updated by the algorithm itself. The two inputs of each node i are the value to aggregate (v_i) and the current set of neighbors (n_i) as given by the failure detector.

The state of each node i consists of a mapping F_i from node ids to flows; it stores, for each current neighbor, the flow along the edge to that node.

The message-generation function takes the state (the flows) and a neighbor id, and returns the message to be sent to that node. Every round, all nodes send messages to all of their neighbors. A single type of message is sent, containing the self id i, the flow $F_i(j)$ to the respective neighbor j, and the aggregate estimate (line 7). When

no flow value is available for a given neighbor, initially or when a new node starts participating, the value 0 is used. We assume for notational convenience that applying a mapping M to a non-mapped key k yields 0, i.e., $M(k) = 0$. The estimate is computed by making use of the *estimation function* (line 19), a function of the input value and the flows (Equation 1).

The state-transition function (lines 9–17) 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'. We make use of some auxiliary variables to compute the new state: the flows F updated according to the messages received, and the estimates E (last received) used to compute the new average a. In more detail:

- F is a mapping from current neighbor ids to: the symmetric of the flow in messages, for those neighbors whose messages arrived (line 11); the current value, if any, in the case of message loss (line 12).
- 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 13); for neighbors whose messages arrived, the estimate sent (line 14); otherwise, the estimate according to the estimation function, using the flows at the beginning of the round (this is the estimate sent to all neighbors at the beginning of the round) (line 15).
- a is simply the average of the estimates in the mapping E, and represents the new estimate towards which the node will lead its neighbors to converge in the next round (line 16).

Finally, the new mapping F_i' is calculated by adjusting each flow in F so that the estimates move towards a: the estimate for node i in the beginning of next round will be a; each neighbor would compute a as its estimate at the end of the next round if it did not receive other messages from its own neighbors (e.g., if it has node i as its only neighbor).

3.2. Algorithm – Asynchronous Version

The first description of Flow Updating used the synchronous network model. Such model is useful, to reason only in the number of communication rounds (while taking into account concurrent execution), and therefore, useful to perform a quantitative evaluation, namely to compare the performance of different algorithms without having to make assumptions about latencies. Real distributed systems, however, give weaker timing guaranties, and the issue of how to practically implement Flow Updating arises.

As we are talking about a fault-tolerant algorithm, namely that works under message loss, the classic technique of using a synchronizer [28] to allow a synchronous algorithm to be transformed into an asynchronous one cannot be applied. Moreover, even though it is advantageous (as we will see in the evaluation) to compute an average after having collected contributions from many neighbors, lockstep computation as in the synchronous version is not an inherent requirement of flow updating.

These considerations lead us to present flow updating for the asynchronous network model (see e.g., Chapters 8 and 14 of [27]) directly in two variants. The first one is the more natural in the asynchronous setting, in which the algorithm reacts to each message, performing a pairwise averaging of two values; the second mimics the spirit of the synchronous one, and tries to collect messages from all neighbors before performing an averaging step.

The pairwise asynchronous version of Flow Updating is presented in Algorithm 2, while the collect-all version is presented in Algorithm 3. The algorithms have inputs which are not only read but can change arbitrarily and continuously due to external factors, state variables which are manipulated by the algorithm, and events to which the algorithms react, namely: init_i when node i starts computing, $\text{receive}_{j,i}$ representing node i receiving a message sent by node j, and tick_i representing a clock tick at node i. A node i reacts to an event by changing local state and possibly sending messages to neighbors, through $\text{send}_{i,j}$.

inputs:
  v_i, value to aggregate
  n_i, set of neighbors given by failure detector
  t_i, timeout value (in number of ticks)

state:
  F_i, flows: initially, $F_i = \{\}$
  E_i, neighbors estimates: initially, $E_i = \{\}$
  C_i, ticks since last averaging, for each neighbor: initially, $C_i = \{\}$

on $\text{init}_i()$
  foreach $j \in n_i$ **do**
    $\text{send}_{i,j}(0, v_i)$

on $\text{receive}_{j,i}(f, e)$
  $E_i(j) := e$
  $F_i(j) := -f$
  averageAndSend(j)

on $\text{tick}_i()$
  foreach $j \in n_i$ **do**
    $C_i(j) := C_i(j) + 1$
    if $C_i(j) \geq t_i$ **then** averageAndSend(j)

procedure averageAndSend(j)
  $e = v_i - \sum_{j \in n_i} F_i(j)$
  $a = (E_i(j) + e)/2$
  $F_i(j) := F_i(j) + a - E_i(j)$
  $E_i(j) := a$
  $C_i(j) := 0$
  $\text{send}_{i,j}(F_i(j), a)$

Algorithm 2: Flow Updating algorithm for dynamic networks in the asynchronous network model, pairwise version.

Common behavior upon receive and tick events is factored out in the "averageAndSend" procedure. As before, we assume for notational convenience that applying a mapping M to a non-mapped key k yields 0, i.e., $M(k) = 0$. We also update individual keys through a simple assignment, i.e., $M(k) := x$, regardless of whether they were already mapped.

In both versions each node starts by sending a message to each neighbor. In the pairwise version, the basic behavior, if no failures occurred and the timeout mechanism did not exist, is for each node to "reply" back with a new flow and estimate, after averaging, as soon as it receives new knowledge from a given node. The idea is that it results in communication along different links to adapt to the respective link latency, with messages bouncing back and forth at different rates for different links, without forcing a global lockstep. The second version, by making each node wait for all neighbors before averaging, would behave as the synchronous algorithm under no message failures and no timeout mechanism, i.e., it would force a global lockstep.

inputs:
  v_i, value to aggregate
  n_i, set of neighbors given by failure detector
  t_i, timeout value (in number of ticks)

state:
  F_i, flows: initially, $F_i = \{\}$
  E_i, neighbors estimates: initially, $E_i = \{\}$
  M_i, neighbor ids of messages received since last averaging: initially, $M_i = \{\}$
  c_i, ticks since last averaging: initially, $c_i = 0$

on $\text{init}_i()$
  foreach $j \in n_i$ **do**
    $\text{send}_{i,j}(0, v_i)$

on $\text{receive}_{j,i}(f, e)$
  $E_i(j) := e$
  $F_i(j) := -f$
  $M_i := M_i \cup \{j\}$
  if $M_i \supseteq n_i$ **then** averageAndSend()

on $\text{tick}_i()$
  $c_i := c_i + 1$
  if $c_i \geq t_i$ **then** averageAndSend()

procedure averageAndSend(j)
  $e = v_i - \sum_{j \in n_i} F_i(j)$
  $a = (e + \sum_{j \in n_i} E_i(j))/(|n_i| + 1)$
  foreach $j \in n_i$ **do**
    $F_i(j) := F_i(j) + a - E_i(j)$
    $E_i(j) := a$
    $\text{send}_{i,j}(F_i(j), a)$
  $M_i := \{\}$
  $c_i := 0$

Algorithm 3: Flow Updating algorithm for dynamic networks in the asynchronous network model, collect all version.

In both versions tolerance to message loss is obtained through a "timeout" mechanism, which forces the averaging step and message sending if a given number of local clock ticks have occurred since the last averaging step. In the asynchronous model this says nothing about real time,

as a tick is just an event of which we only know that it will occur infinitely often in an infinite trace. We have, however, presented the algorithms this way (instead of using a more vague "periodically") so that they can be more directly translated to practical implementations where a tick will correspond to the physical clock ticking of the computing node.

An interesting difference between both versions is that the pairwise version uses a per-link timeout (as opposed to a single timeout as the other version). This is because otherwise, if each of two linked nodes a and b kept communicating successfully with all their neighbors except b and a respectively, we could have an infinite trace where a single timeout variable was constantly being reset by the successful communications. This would imply that messages would stop being exchanged between a and b after the loss of the initial messages.

Only one flow and estimate is kept per neighbor. In particular, in the collect-all version, if between averaging steps a newer message arrives from a given neighbor it will overwrite the previously stored values. This means that the values used in the averaging may not be the last ones sent, in the general case of non-FIFO channels; this can also occur in the pairwise version. This is not a problem, and no effort was made to enforce FIFO semantics through sequence numbers.

4. Evaluation

In this section, we provide experimental results to evaluate *Flow Updating* under demanding faulty scenarios, with both churn and message loss. We also compare it against existing average based techniques, and take into consideration some practical concerns, like the use of realistic failure detectors and asynchrony, to assess its implementation in real environments.

For this purpose, we use a custom discrete event simulator which allows evaluating both synchronous and asynchronous algorithms, to compute the COUNT aggregation function (determination of network size). The synchronous algorithm is used everywhere, except in the section devoted to asynchrony. The COUNT aggregation function is used in all simulations, unless stated otherwise. Convergence speed depends on the initial data distribution across the network; COUNT represents an extreme scenario where only one node starts with the value 1 and all others with 0. We chose to use this aggregation function for evaluation because it is the one with the worst performance. The algorithm 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.

We consider two different network topologies: random (where all nodes are randomly linked to each other, according to a predefined degree d), and 2D/mesh (geographical networks, with random uniform node placement, where communication links are established according to a predefined radio communication range, an approximation to the topologies occurring in WSN). The results for each scenario are drawn from 30 trials of the execution of the algorithms under identical settings. In each trial different randomly generated networks with the same characteristics (topology, size and average degree) are used.

The main metric used in most simulation scenarios is the *CV(RMSE)* (Coefficient of Variation of the Root Mean Square Error)[4], which expresses the global accuracy reached by an algorithm. This metric allows the analysis of the speed and message load of the tested algorithms, when combined with the proper criteria, respectively: time (or number of rounds) and number of messages sent (by each node). 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[5].

[4] 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: $\frac{1}{\bar{a}}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(e_i-\bar{a})^2}$

[5] As referred in [29], the energy consumed to transmit a single

4.1. Performance Comparison Under no Faults

Here, *Flow Updating* (FU) is compared to three significant distributed aggregation algorithms from the same class (i.e., averaging): Push-Sum Protocol (PSP) [18], Push-Pull Gossiping (PPG) [7], and Distributed Random Grouping (DRG) [19]. 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 grant them the best performance in each simulated scenario (e.g., the probability to become leader in DRG).

A comparison with the recent Limosense [23] approach was tried but was not viable, since simulations with concurrent executions quickly lead to runs where the algorithm crashes due to divisions by 0. The root cause in the algorithm formulation is probably quite addressable but it precluded a direct comparison here. However, since Limosense inherits the convergence behavior of PSP our comparison with that protocol provides a suitable reference in the no faults scenario.

The comparison was performed on fixed and reliable network topologies (i.e. without dynamism nor faults) with the same size $n = 1000$, but two different average connection degrees, i.e. $d \approx 3$ and $d \approx 10$. The results are depicted by Figures 1 and 2 for random networks, and Figures 3 and 4 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 and more details can be found in [21].

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 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). 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 network (which more closely corresponds to WSN), FU considerably outperforms the other techniques.

It was also observed in [24] that in large hypercube topologies FU exhibits a worst performance than PSP. In this type of topology, and in other networks with high connection degree, it is possible to improve the performance of FU by applying simple heuristics to use a subset of the available neighbors for the aggregation process (i.e., ignoring some links) as shown in [30]. A detailed study of the performance issues in these scenarios and heuristics for their improvement is left for future work.

4.2. Churn and Message Loss

We now consider the COUNT aggregate computation in dynamic settings. Computing the COUNT aggregate is particularly demanding, and useful, in networks where the number of nodes is actively changing. First, we will consider this task in the absence of message loss and later introduce that additional perturbation.

All networks considered in every churn scenario start with the same size ($n = 1000$), and the same approximated average connection degree $d \approx \log n$ (where log is the natural logarithm). The choice of d was influenced by [31], where it is stated that some nodes 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 value used was enough to avoid network partitioning for the simulated churn scenarios (e.g., failure of one quarter of the nodes).

We start by considering a random network scenario, when subject to both drastic and continuous changes of the network membership. For this purpose, we succes-

bit corresponds roughly to the one required to execute thousands of instructions.

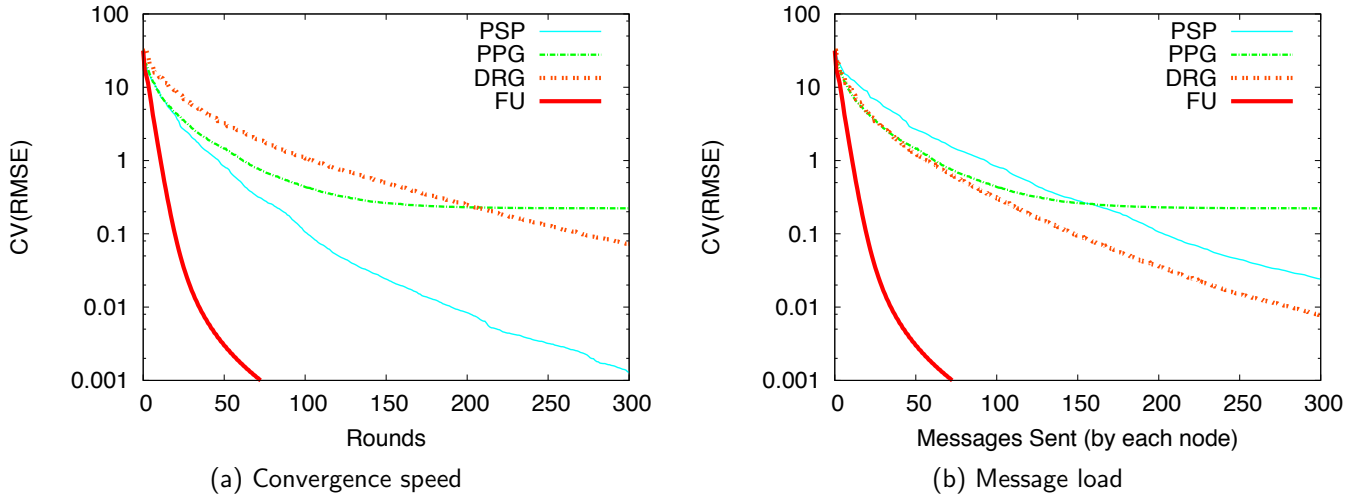


Figure 1: Comparison of averaging algorithms, on random networks with $n = 1000$ and $d \approx 3$.

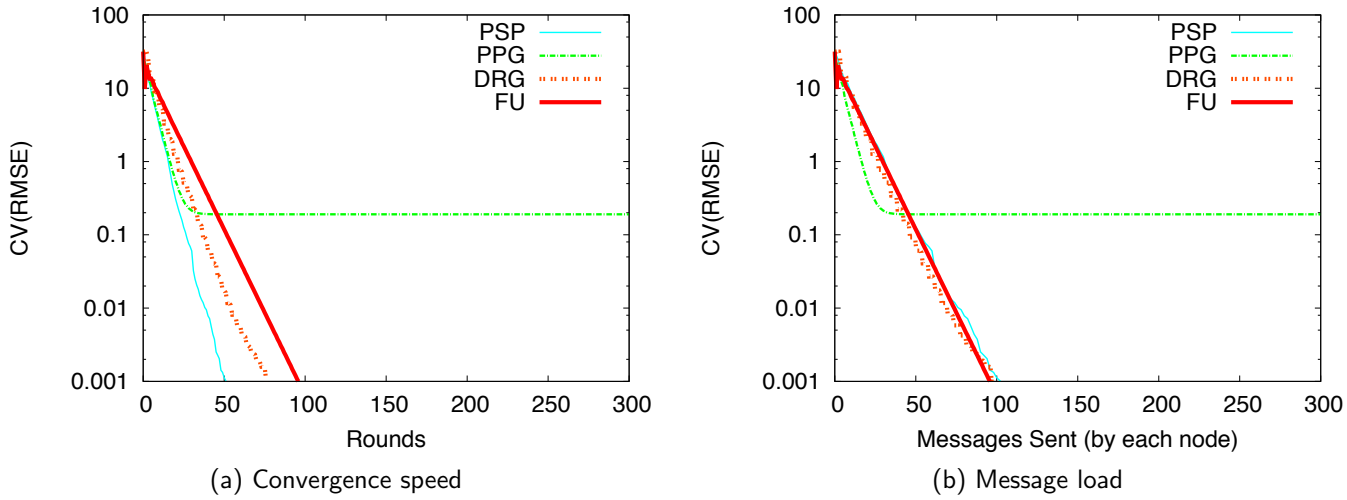


Figure 2: Comparison of averaging algorithms, on random networks with $n = 1000$ and $d \approx 10$.

sively applied a sudden departure (catastrophic crash) and arrival of 25% of the initial nodes, followed by an arrival and departure of the same portion of nodes at a constant rate (10 nodes per round). For a matter of clarity, a stability period of 50 rounds is introduced between each network change.

First, we compare *Flow Updating* (FU) with *Push-Pull Gossiping*[7] (PPG), and *Push-Pull Ordered Wait* [21] (PPOW is a fix of PPG that solves its atomicity problems), in the described random dynamic network scenario without message loss. PPG implements a restart mechanism to cope with churn, starting a new instance of the algorithm after a predefined number of rounds (epoch), and prevents new nodes from participating in the current epoch. Similarly to PPG, PPOW was extended with a restart mechanism, but instead of delaying the participation of new nodes to the next epoch, joining nodes are allowed to participate immediately. This modification was applied since it yielded more favorable results to PPOW in all performed experiments.

Figure 5 shows the results obtained, using an epoch length of 50 rounds. 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),

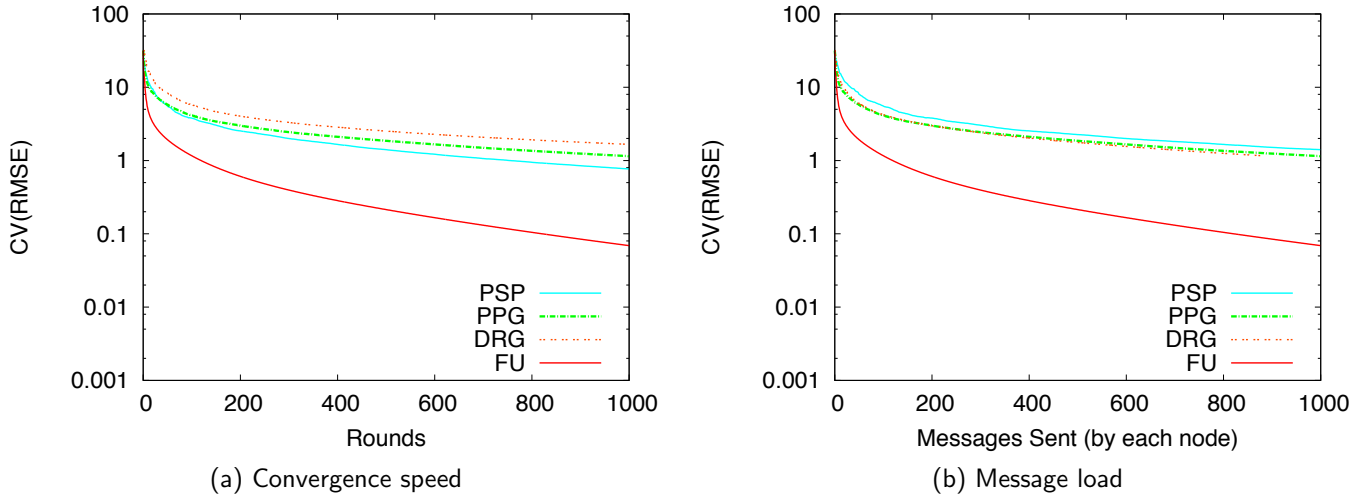


Figure 3: Comparison of averaging algorithms, on 2D/mesh networks with $n = 1000$ and $d \approx 3$.

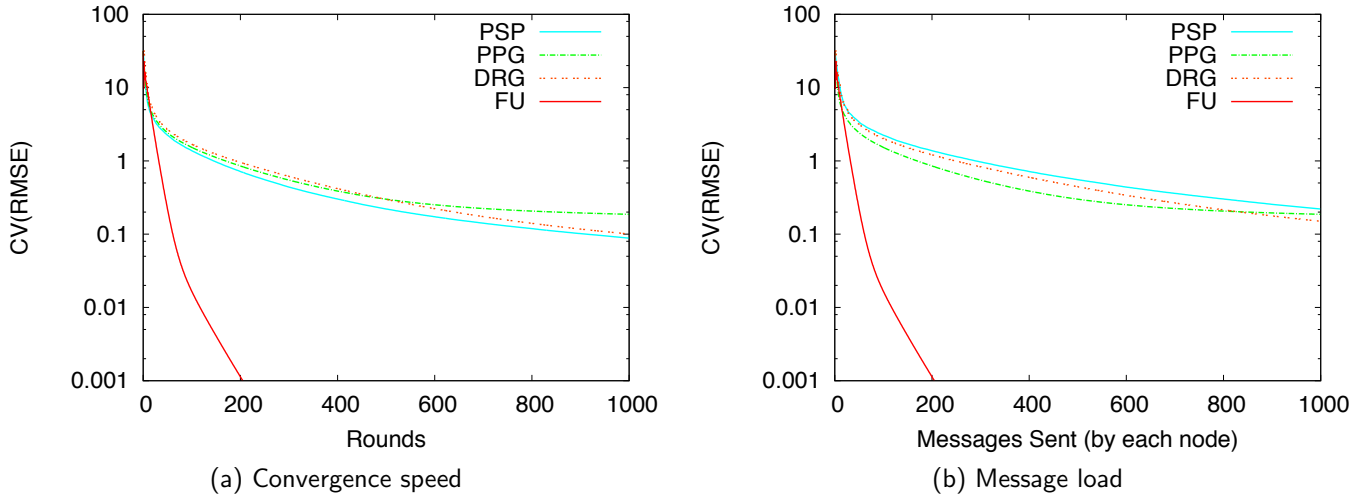


Figure 4: Comparison of averaging algorithms, on 2D/mesh networks with $n = 1000$ and $d \approx 10$.

which is solved by PPOW that converges to the expected value. More importantly, these results expose the effect of the restart mechanism, that introduces an undesirable delay to respond to network change. This delay is also observed even if only the estimate at the end of each epoch is considered 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 restart mechanism introduces a trade-off between the response time to network change 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. FU clearly outperforms the other approaches (PPG and PPOW) which are unable to adapt to network changes. For this reason, the remainder of the evaluation focuses exclusively on Flow Updating.

We now evaluate the behavior of FU on the same dynamic random network. But besides churn, we also consider that each individual message can be lost according to a given probability. Figure 6 shows more clearly that

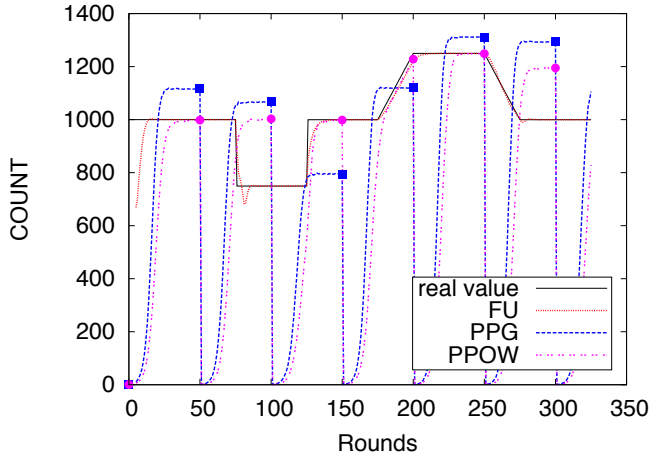


Figure 5: Comparison of Flow Updating (FU), Push-Pull Gossiping (PPG) and Push-Pull Ordered Wait (PPOW): Average of estimates in a dynamic random network, with no message loss.

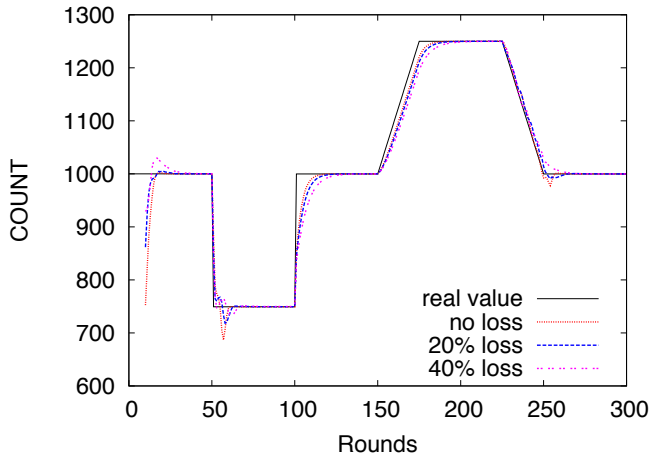


Figure 6: Average of estimates in a dynamic random network with message loss.

the mean of the estimates produced by FU closely follows the network changes.

Figure 7 shows the CV(RMSE) over time, allowing the observation of the global accuracy variation due to network dynamism. This metric compares each individual estimate against the actual network size, as perceived by an external observer that can inspect the whole network in 0 rounds. This is a very demanding metric, since in any actual distributed algorithm nodes would have a delay proportional to diameter rounds before knowing the network size.

From Figures 6 and 7, we observe that even considerable message loss (20% and 40%) only slightly affects

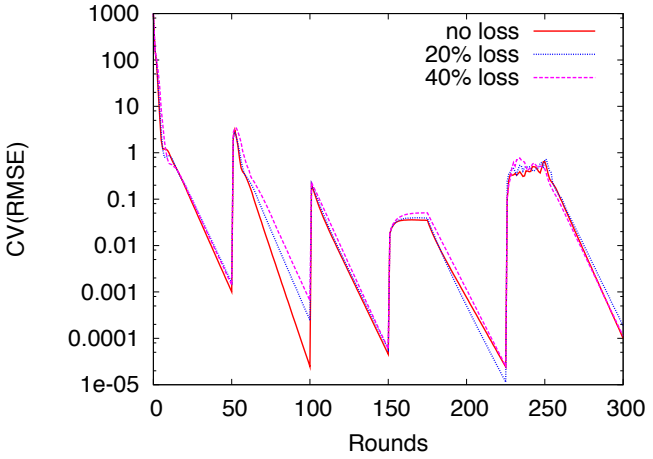


Figure 7: Coefficient of variation of the RMSE in a dynamic random network with message loss.

convergence speed and the ability of the algorithm to cope with churn. Curiously, in some situations the algorithm even benefits from message loss, increasing its convergence speed (e.g., 20% loss in rounds 175 to 225 being better than no loss). We found out that it is possible to increase convergence speed by "deactivating" some communication links [30]. 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.

The results confirm the fast convergence of the algorithm during stability 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 this particular case, for the considered churn rate (10 nodes per round), the arrival 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 the impact on node estimation accuracy. The effect of churn on each node estimate is clearly

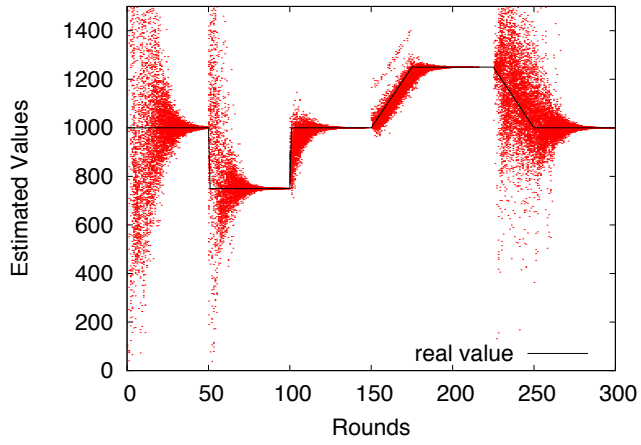


Figure 8: Estimates distribution in a dynamic random network with 20% of message loss.

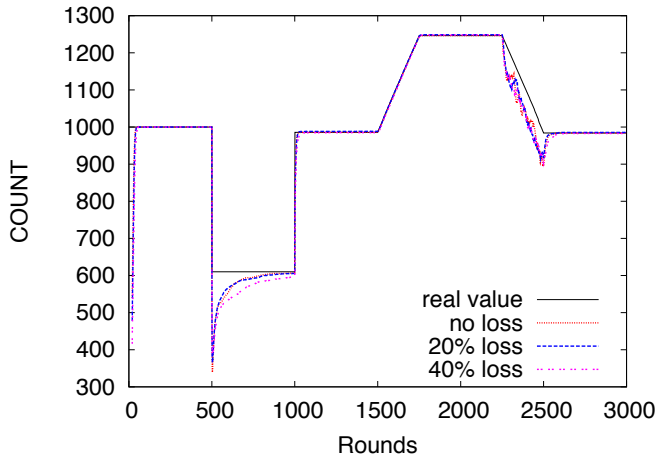


Figure 9: Average of estimates in a dynamic 2D/mesh network with message loss.

depicted by Figure 8, which shows the distribution of individual estimates along time, considering 20% of message loss[6].

The previous simulation scenarios are now applied using 2D/mesh network topologies. Since the convergence speed of these kind of networks is much slower, a bigger stability period (500 rounds) and slower churn rate (1 node per round) were considered. Results are presented in Figures 9 and 10. In these settings, the behavior of *Flow Updating* is similar to the one previously described for random networks, although a deeper contrast between the effect of node arrival and departure is observed. Namely,

[6]The graphic of the distribution of node estimates in a scenario without message loss is very similar to the case of 20% faults.

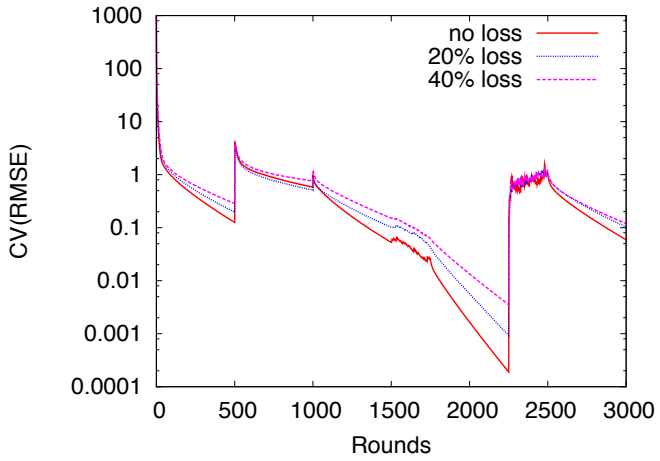


Figure 10: Coefficient of variation of the RMSE in a dynamic 2D/mesh network with message loss.

the perturbation introduced by a sudden (round 1000) or continuous (rounds 1500 to 1750) arrival of nodes is very small. On the contrary, node departure/crash has a greater impact in this kind of network.

Node departure/crash breaks the flows established between nodes, and can result in the removal of links connecting different clusters, breaking the 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, leading to a smaller adjustment of the existing flows in order to converge to the new aggregate.

4.3. Failure Detection

Failure Detectors (FD) are oracles providing information about whether processes have failed [26]; however, they do not necessarily provide correct information. 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. Here, the impact of realistic unreliable FD in the execution of FU on dynamic settings is evaluated.

Practical implementations of FD are commonly timeout-based [32]. 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 (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. Figure 11 shows how the performance of FU is affected by FD with different timeout values, when subjected to churn and message loss.

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 11(a) and 11(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 may cause incorrect suspicion of some nodes, making FU remove the flows of a correct process, and the whole system to 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 towards the correct result. However, since message loss occurs continuously over time, this situation will 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 towards the correct value.

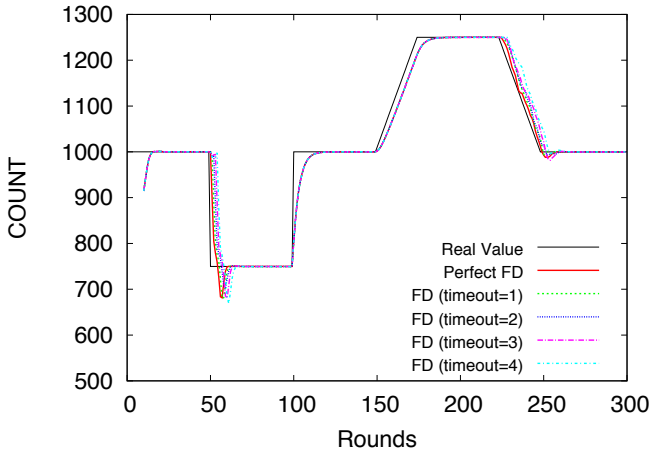
As shown by Figures 11(c)–11(d) and 11(e)–11(f), the higher the amount of message loss the higher the impact on FU, especially when using a FD with a small timeout. This is because FD with small timeout values only requires a few consecutive message losses to incorrectly mark a node as suspected (e.g., only one message loss is enough for the FD with timeout 1), while FD with larger 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. Therefore, it is more appropriate to use conservative FD timeouts.

The results obtained show that the selection of an appropriate FD timeout is fundamental to ensure a good performance and accuracy of FU. It is important to use a practical FD that minimizes the number of incorrect suspicions, in order to avoid an undesired performance degradation of FU. Despite the additional delay introduced by a conservative FD to react to network changes (i.e., node departure/crash), this kind of FD should, therefore, be preferred. This recommendation is valid for any network topology, as it is expected that fault detection will affect FU in the same way.

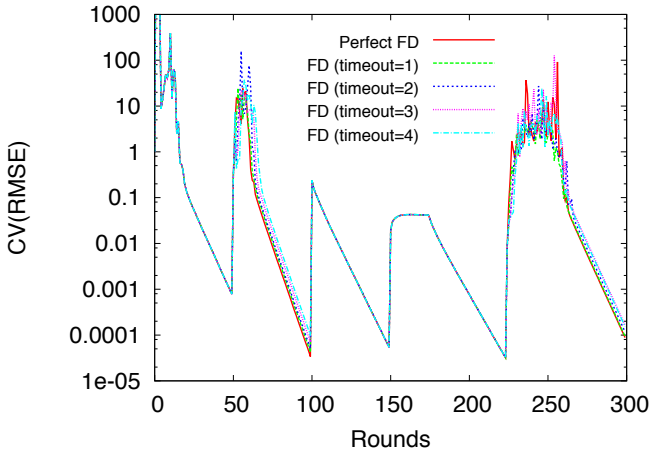
4.4. Input Value Change

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; after 50 rounds 50% of the nodes (randomly chosen) start increasing their input value 5% each round, during 50 rounds; finally, they 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 hypothetical 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 $d \approx 3$).

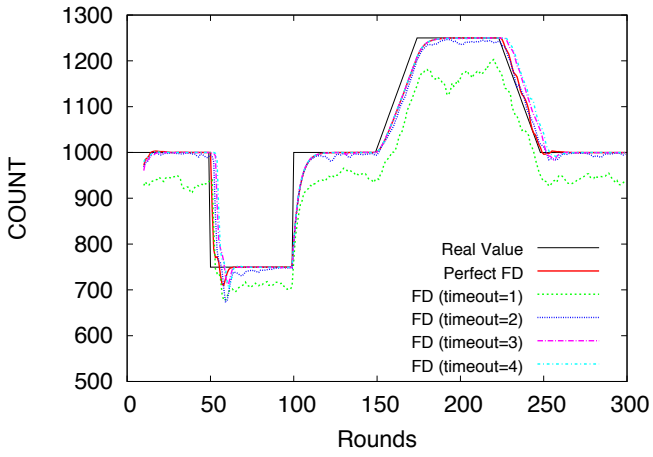
Figure 12 shows that the average of the estimates produced by all nodes closely follows the change of the global



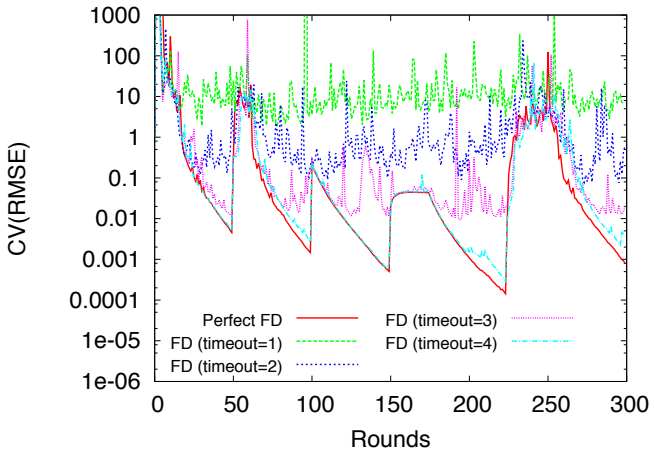
(a) Average estimate (no loss)



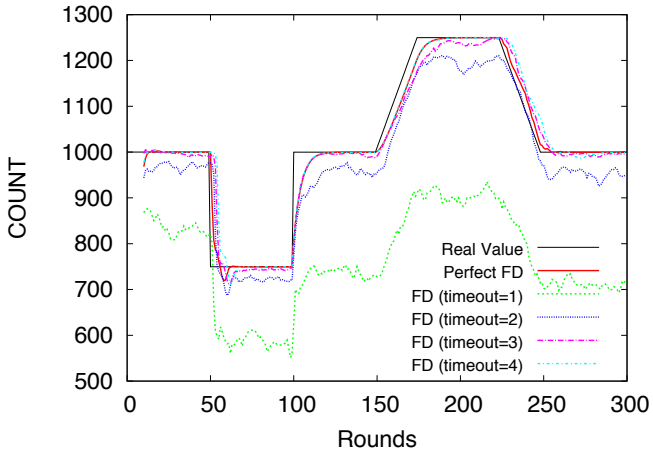
(b) Convergence speed (no loss)



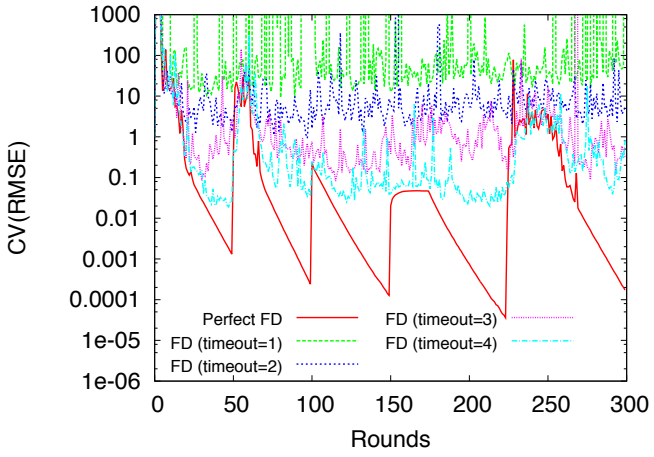
(c) Average estimate (10% loss)



(d) Convergence Speed (10% loss)



(e) Average estimate (20% loss)



(f) Convergence Speed (20% loss)

Figure 11: Effect of FD on the execution of FU in dynamic settings with message loss – random networks ($n = 1000$, $d \approx \log n$).

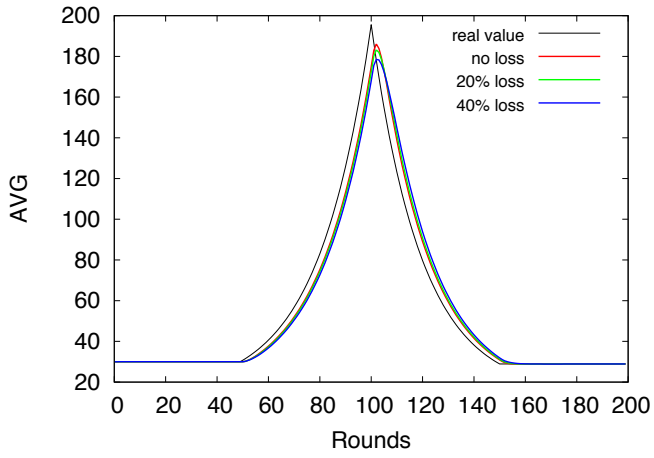


Figure 12: Average of estimates in a random network with input value changes and message loss.

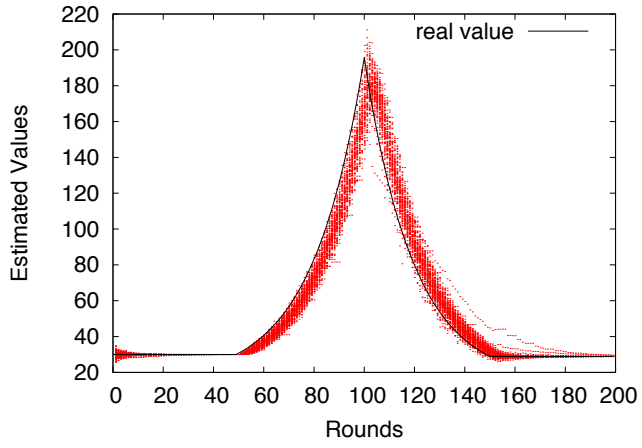


Figure 13: Estimates distribution in a random network with input value changes and 20% message loss.

average (with a small delay), even under considerable message loss. A more precise view of the estimate of all nodes over time is given by Figure 13, for a simulation with 20% 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 graphs for the scenarios with 0% and 40% message loss (not shown) are very similar. 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.

4.5. Asynchrony

We now evaluate the asynchronous version of FU, when used in realistic settings. The algorithm was described for asynchronous networks, working under all timing assumptions. For evaluation purposes, given that there is no need for a global clock, that processing time is negligible, and assuming that local clock drift will also be negligible, we focus on the effect of variable latency in communication, assuming that a practical implementation will have the timeout variable reflecting "elapsed real time".

The evaluation aims to answer two questions: 1) which of the two asynchronous versions of FU is preferable in practice; 2) how should the timeout be chosen, for a given latency distribution. As before, the criteria used are convergence speed and number of messages sent.

The transmission time of each message was chosen according to a fixed probability distribution, with no attempt to distinguish different links. In particular, a rough approximation to the distribution of message latencies observed in PlanetLab [33] was defined, according to the RTT (Round Trip Time) measurements presented in [34]. Inspired by [35], we approximated transmission times with the sum of two components: a queuing delay given by a Weibull distribution, and a minimum transmission delay. More precisely, 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, resulting in a distribution with an average of 89 ms and with most of the transmission times below 150 ms, as presented in Figure 14.

Timeout values of 25, 50, 100, 125, 150, and 300 ms were considered to evaluate the performance of both asynchronous versions of FU, under 20% message loss. The results are presented in Figures 15 and 16.

The first conclusion that can be reached is that very small timeouts are not worthwhile, because a small improvement in convergence speed is paid by a significant increase of messages transmitted.

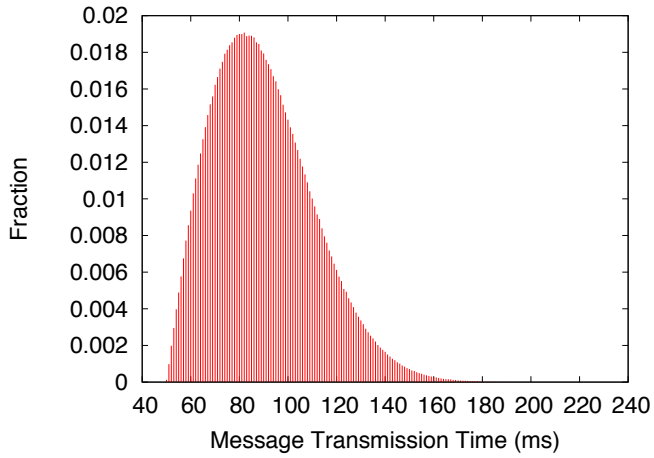


Figure 14: Distribution of message latencies.

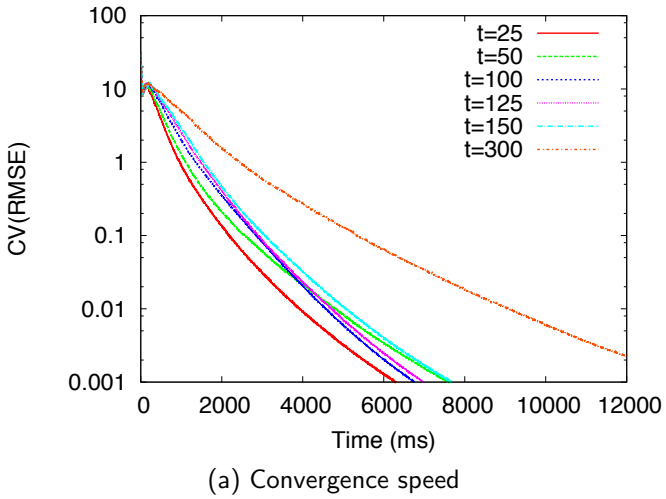
Comparing both asynchronous versions, we can see that the version that mimics the synchronous one and waits for messages from all neighbors is preferable. The pairwise version, even though slightly faster, pays a high price in messages transmitted: the pairwise version can be around 30% faster but at a cost of sending from 3 to more than 10 times as many messages.

Considering the choice of timeout, we can see that sensible choices will be values above the average message latency, in a high percentile position in the distribution, and that there is a trade-off between convergence speed and messages transmitted; the appropriate choice depends on the objective pursued.

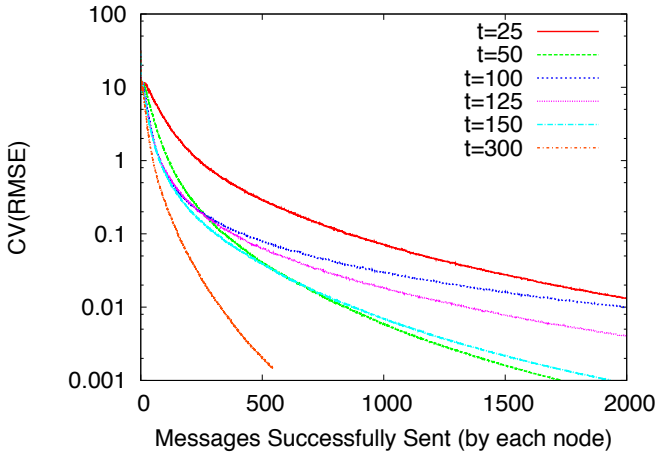
5. Conclusions

Average-based approaches constitute an important segment of aggregation algorithms due to their independence from network topology and convergence to any desired precision. Our previous works on averaging by Flow Updating, already introduced fault tolerance and dynamism: in static settings, achieving up to an order of magnitude improvement in convergence speed without increasing the message load [9]; self-adapting to network changes even with high levels of message loss [10].

Here we have considered relevant practical concerns, like the use of unreliable failure detectors and introduced



(a) Convergence speed

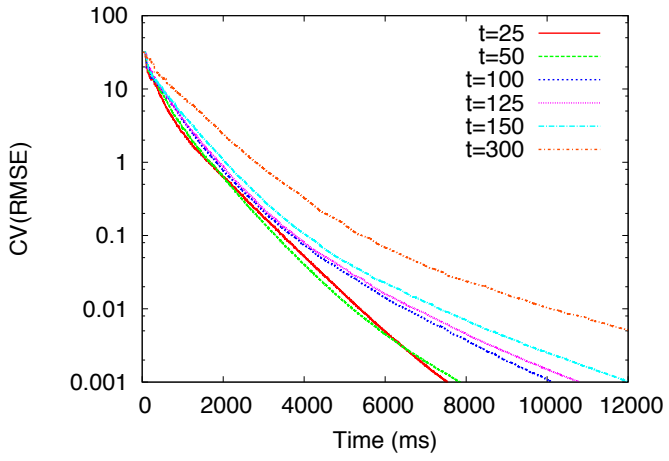


(b) Message load

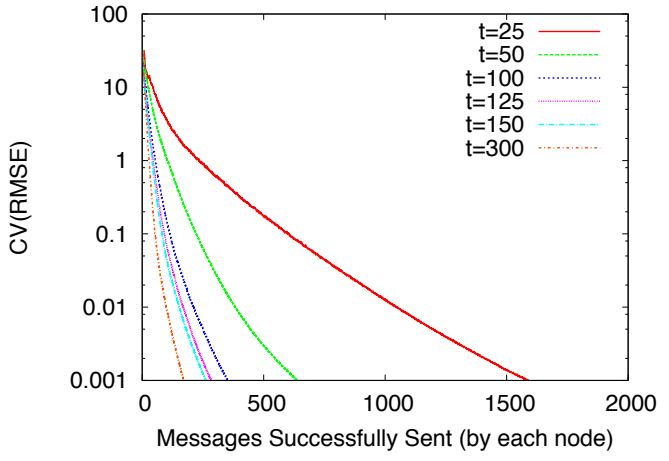
Figure 15: Execution of asynchronous pairwise version on random networks with 20% of message loss.

and evaluated an asynchronous version, showing that Flow Updating can be effectively applied in real environments. We have also brought attention to vulnerabilities of popular averaging techniques when faced with failures and dynamic environments. It is our belief that these shortcomings are not easy to fix and that "mass exchange" must give way to idempotent flow management, in order to address these demanding scenarios.

Flow Updating uses a simple strategy to support dynamism, where entries for neighbor nodes are added or removed according to the current participants (given by a realistic failure detector implementation). This simple design adapts to abrupt changes of network membership



(a) Convergence speed



(b) Message load

Figure 16: Execution of asynchronous collect-all version on random networks with 20% of message loss.

and tracks continuous variations of network size.

Evaluation showed that Flow Updating clearly outperforms previous strategies; unlike most, it adapts in a continuous fashion without requiring protocol restarts.

It allows all nodes to continuously adjust their estimates according to network changes (i.e., churn) and input values change, quickly converging to the current network average, even with very high levels of message loss.

Finally, we have shown that Flow Updating can be successfully applied in practice, relying on realistic failure detector implementations, and using a simple timeout strategy to operate in asynchronous settings with unbounded communication latency. In particular, failure detectors that reduce the number of incorrect suspicions should be preferred (i.e., conservative timeout-based implementations). In asynchronous settings, a better performance is achieved by an algorithm that mimics the synchronous one, collecting and averaging values from all neighbors (as opposed to reacting to individual messages), and using timeout values in a high percentile of the underlying message latency distribution.

Acknowledgment

This work was partially funded by FCT PhD grant SFRH/BD/33232/2007 and by project Norte-01-0124-FEDER-000058, co-financed by the North Portugal Regional Operational Program (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF).

References

[1] R. V. Renesse, The importance of aggregation, Future Directions in Distributed Computing 2584 (2003) 87–92.

[2] M. Jelasity, A. Montresor, Epidemic-style proactive aggregation in large overlay networks, in: Proc. 24th International Conference on Distributed Computing Systems, 2004, pp. 102–109.

[3] S. Madden, M. Franklin, J. Hellerstein, W. Hong, TAG: a Tiny AGgregation service for ad-hoc sensor networks, ACM SIGOPS Operating Systems Review 36 (SI) (2002) 131–146.

[4] J. Li, K. Sollins, D. Lim, Implementing aggregation and broadcast over distributed hash tables, ACM SIGCOMM Computer Communication Review 35 (1) (2005) 81–92.

[5] A. Ganesh, A. Kermarrec, E. L. Merrer, L. Massoulié, Peer counting and sampling in overlay networks based on random walks, Distributed Computing 20 (4) (2007) 267–278.

[6] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, A. Demers, Decentralized schemes for size estimation in large and dynamic groups, in: Proc. 4th IEEE International Symposium on Network Computing and Applications, 2005, pp. 41–48.

[7] M. Jelasity, A. Montresor, O. Babaoglu, Gossip-based aggregation in large dynamic networks, ACM Transactions on Computer Systems (TOCS) 23 (3) (2005) 219–252.

[8] O. Kennedy, C. Koch, A. Demers, Dynamic approaches to in-network aggregation, in: Proc. 25th IEEE International Conference on Data Engineering (ICDE), 2009, pp. 1331–1334.

[9] P. Jesus, C. Baquero, P. S. Almeida, Fault-tolerant aggregation by flow updating, in: Proc. 9th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Vol. 5523 of Lecture Notes in Computer Science, Springer, Lisbon, Portugal, 2009, pp. 73–86.

[10] P. Jesus, C. Baquero, P. S. Almeida, Fault-Tolerant Aggregation for Dynamic Networks, in: 29th IEEE Symposium on Reliable Distributed Systems, 2010, pp. 37–43.

[11] S. Madden, R. Szewczyk, M. Franklin, D. Culler, Supporting aggregate queries over ad-hoc wireless sensor networks, in: Proc. 4th IEEE Workshop on Mobile Computing Systems and Applications, 2002, pp. 49–58.

[12] Y. Birk, I. Keidar, L. Liss, A. Schuster, R. Wolff, Veracity radius: capturing the locality of distributed computations, in: Proc. 25th annual ACM symposium on Principles of Distributed Computing (PODC), 2006, pp. 102–111.

[13] Y. Sun, H. Luo, S. K. Das, A trust-based framework for fault-tolerant data aggregation in wireless multimedia sensor networks, Dependable and Secure Computing, IEEE Transactions on 9 (6) (2012) 785–797. doi:10.1109/TDSC.2012.68.

[14] C. Baquero, P. Almeida, R. Menezes, P. Jesus, Extrema propagation: Fast distributed estimation of sums and network sizes, Parallel and Distributed Systems, IEEE Transactions on 23 (4) (2012) 668–675. doi:10.1109/TPDS.2011.209.

[15] D. Mosk-Aoyama, D. Shah, Computing separable functions via gossip, in: Proc. 25th annual ACM symposium on Principles of Distributed Computing (PODC), 2006, pp. 113–122.

[16] L. Massoulié, E. Merrer, A.-M. Kermarrec, A. Ganesh, Peer counting and sampling in overlay networks: random walk methods, in: Proc. 25th annual ACM symposium on Principles of Distributed Computing (PODC), 2006, pp. 123–132.

[17] S. Mane, S. Mopuru, K. Mehra, J. Srivastava, Network size estimation in a peer-to-peer network, Tech. rep., University of Minnesota, Department of Computer Science (Sep. 2005).

[18] D. Kempe, A. Dobra, J. Gehrke, Gossip-based computation of aggregate information, in: Proc. 44th Annual IEEE Symposium on Foundations of Computer Science, 2003, pp. 482–491.

[19] J.-Y. Chen, G. Pandurangan, D. Xu, Robust computation of aggregates in wireless sensor networks: Distributed randomized algorithms and analysis, IEEE Trans. Parallel Distrib. Syst. 17 (9) (2006) 987–1000.

[20] F. Wuhib, M. Dam, R. Stadler, A. Clemm, Robust monitoring of network-wide aggregates through gossiping, in: Proc. 10th IFIP/IEEE International Symposium on Integrated Network Management, 2007, pp. 226–235.

[21] P. Jesus, C. Baquero, P. S. Almeida, Dependability in aggregation by averaging, in: Proc. Simpósio de Informática (INForum), Lisboa, Portugal, 2009, pp. 457–470.

[22] P. S. Almeida, C. Baquero, M. Farach-Colton, P. Jesus, M. A. Mosteiro, Fault-tolerant aggregation: Flow-updating meets mass-distribution, in: A. F. Anta, G. Lipari, M. Roy (Eds.), OPODIS, Vol. 7109 of Lecture Notes in Computer Science, Springer, 2011, pp. 513–527.

[23] I. Eyal, I. Keidar, R. Rom, Limosense – live monitoring in dynamic sensor networks, in: Algorithms for Sensor Systems, Vol. 7111 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2012, pp. 72–85.

[24] W. N. Gansterer, G. Niederbrucker, H. Straková, S. S. Grotthoff, Scalable and fault tolerant orthogonalization based on randomized distributed data aggregation, Journal of Computational Science 4 (6) (2013) 480–488. doi:10.1016/j.jocs.2013.01.006.

[25] P. Jesus, C. Baquero, P. S. Almeida, A Survey of Distributed Data Aggregation Algorithms, IEEE Communications Surveys and Tutorials (PrePrint). doi:10.1109/COMST.2014.2354398.

[26] T. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of the ACM (JACM 43 (2) (1996) 225–267.

[27] N. A. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers Inc., 1996.

[28] B. Awerbuch, Complexity of network synchronization, J. ACM 32 (4) (1985) 804–823.

[29] S. Aslam, F. Farooq, S. Sarwar, Power consumption in wireless sensor networks, in: Proceedings of the 7th International Conference on Frontiers of Information Technology (FIT), Punjab University College of Information Technology (PUCIT), University of the Punjab, Anarkali, Lahore, Pakistan, Abbottabad, Pakistan, 2009, pp. 14:1–14:9.

[30] P. Jesus, C. Baquero, P. S. Almeida, Using less links to improve fault-tolerant aggregation, 4th Latin-American Symposium on Dependable Computing (LADC), [Fast Abstract] (2009).

[31] M. F. Kaashoek, D. R. Karger, Koorde: A simple degree-optimal distributed hash table, in: Proc. 2nd International Workshop on Peer-to-Peer Systems (IPTPS), Vol. 2735 of Lecture Notes in Computer Science, Springer, 2003, pp. 98–107.

[32] M. Dixit, A. Casimiro, Adaptare-FD: A Dependability-Oriented Adaptive Failure Detector, in: 29th IEEE Symposium on Reliable Distributed Systems, 2010, pp. 141–147.

[33] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, M. Bowman, PlanetLab: An Overlay Testbed for Broad-Coverage Services, ACM SIGCOMM Computer Communication Review 33 (3) (2003) 3–12.

[34] L. Tang, Y. Chen, F. Li, H. Zhang, J. Li, Empirical Study on the Evolution of PlanetLab, in: Proceedings of the 6th International Conference on Networking (ICN), IEEE, 2007, pp. 64–70.

[35] J. Hernandez, I. Phillips, Weibull mixture model to characterise

end-to-end Internet delay at coarse time-scales, IEE Proceedings - Communications 153 (2) (2006) 295–304.