Robust Gossip-Based Aggregation: A Practical Point of View

Gerhard Niederbrucker* Wilfried N. Gansterer*

Abstract

Over the last years, several gossip-based aggregation algorithms have been developed which focus on providing resilience in failure-prone distributed systems. The main objective of such algorithms is the efficient in-network computation of aggregates even in the case when system failures occur during runtime. In this paper, we evaluate performance and limitations in practical computations of those gossip-based aggregation algorithms with the most promising theoretical fault tolerance properties.

Theoretical analyses of these algorithms usually address only the principal ability of handling or overcoming a certain kind of system failure. Most of the time, there are no formal results on the concrete impact of failure handling on the performance of the algorithms, e. g., in terms of convergence speed. This leaves a wide gap between theory and practice, as we illustrate in this paper. In order to bridge this gap, we first categorize common system failures of interest. Then, we experimentally investigate how well these common failure types are handled in practice by the considered algorithms and up to which extent these state-of-the-art methods provide a reasonable degree of fault tolerance in practice. Our experimental studies reveal (*i*) that certain failure handling approaches which work in theory exhibit unacceptable performance in practice and (*ii*) that in some cases the failure handling mechanisms used introduce new problems, e. g., numerical inaccuracy.

Our investigations illustrate that for some failure types (such as permanent node failures) further algorithmic advances are required to achieve resilience with a reasonably small overhead and acceptable performance.

1 Introduction

Gossiping or *epidemic* protocols can be used for computing aggregates of data across distributed systems. Gossip-based algorithms provide high flexibility with respect to the underlying computing infrastructure, scale well due to their focus on nearest neighbor communication and promise a high potential for providing fault tolerance at the algorithmic level. Recent approaches aim at utilizing this potential by integrating fault tolerance mechanisms of different kinds and nature into less resilient base methods. This integration of fault tolerance mechanisms causes usually only minor overhead and the resulting algorithms are *theoretically* able to handle a wide range of system failures (see Section 2 for a comprehensive discussion of the algorithms we consider in this paper). While in theoretical investigations usually only the principal ability of overcoming system failures is studied, the concrete impact of a failure handling on the overall computation, e. g., in terms of convergence speed and performance, is usually not discussed. Moreover, practical aspects like the influence of floating-point arithmetic on the theoretical results are not discussed in the literature, and experimental evaluations are carried out in varying environments and with respect to varying metrics. Therefore, the available experimental results in the literature cannot be compared directly. Thus, we aim for a thorough comparison and an analysis of the practical applicability of recently proposed fault tolerant gossiping algorithms to identify inherent problems and open questions for future research.

In particular, we investigate how well the failure handling mechanisms in current gossip-based aggregation algorithms work in practice. In the course of this, we aim for a comparison of the strengths and weaknesses of existing algorithms as well as for an analysis how their theoretically predicted behavior is affected in practice. In this paper, we consider the recently proposed fault tolerant aggregation algorithms *Flow-Updating* [1], *LiMoSense* [2], *Push-Flow* [3] and the novel *Push-Cancel-Flow* [4] (cf. Section 2). Moreover, as a baseline for failure-free scenarios, we include the *Push-Sum* algorithm [5] which is closely related to the discussed algorithms, but is much more limited in terms of fault tolerance. Furthermore, we precisely define various classes of failures of interest and simulate them in our experimental evaluations. In order to perform a thorough comparison despite all the randomness present in gossip-based algorithms, we also develop a sound evaluation methodology which produces unbiased results.

*University of Vienna, Research Group Theory and Applications of Algorithms. Email: gerhard.niederbrucker@univie.ac.at, wilfried.gansterer@univie.ac.at

Contributions. We present an in-depth experimental analysis of state-of-the-art fault tolerant gossip-based aggregation algorithms and study their behavior in realistic scenarios which are out of reach of the corresponding theoretical results available in the literature. We observe that the characteristics of real computations and floating-point arithmetic have partly—depending on the concrete algorithm—a major impact on fault tolerance properties. Moreover, we observe that certain fault tolerance mechanisms lead to unexpectedly high overhead and we can even identify commonly shared problems in the considered algorithms. By these observations we bridge the gap between the (partially) existing theoretical analysis of the algorithms and their behavior in practice. Thus, these observations show that the theoretical ability of overcoming system failures can be insufficient in practice. For carrying out our experiments we use a self-made simulator which allows a wide range of experiments up to a large scale.

Synopsis. In Section 2 we review the considered aggregation algorithms and discuss their theoretical properties and expected (fault tolerance) behavior. In Section 3 we summarize our evaluation methodology and precisely define all notions required to empirically verify the expected (fault tolerance) properties of the considered algorithms. Section 4 is devoted to our experimental work and reveals several weaknesses of the considered algorithms which where not identified in the theoretical analyses existing so far. Section 5 concludes the paper.

2 Robust Gossip-Based Aggregation

In this section we clarify which kind of aggregation problems we are going to consider and which types of fault tolerance can be destinguished. Moreover, we briefly discuss the algorithms we are going to evaluate later on.

2.1 Environments and Targets In the following, we always assume a connected, but otherwise arbitrary network of n nodes which allows for point-to-point communication between connected nodes. Formally, the network is modeled by an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where we identify the vertices in the graph (nodes in the network) by natural numbers, i. e., $\mathcal{V} = \{1, \dots, n\}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The neighborhood of node i is defined as $\mathcal{N}_i = \{j \,|\, (i,j) \in \mathcal{E}\}$ and the communication of a node is limited to its neighborhood. Moreover, without loss of generality, we focus our investigations on the problem of *scalar* distributed aggregation. Furthermore, all of the discussed algorithms are easily adapted to perform more complex non-scalar aggregates such as sums/averages of vectors or matrices.

That means, that initially every node i holds an initial value $x_i \in \mathbb{R}$ and the goal is the in-network computation of a global aggregate, e. g., the average $\sum_{i=1}^{n} x_i/n$, locally at each node. Concretely, every node i computes a sequence of estimates $\tilde{a}_i(t)$, $\tilde{a}_i(0) := x_i$ which (theoretically) converges to the target aggregate a (i. e., $\lim_{t\to\infty} \tilde{a}_i(t) = a$) by communicating with its neighborhood only.For reasons which we make clear in the next section, we consider in this paper solely algorithms which aim at robustly computing weighted averages. Thus, all of the discussed algorithms are able to distributedly (locally at each node) compute the global sum and average of the local input data. That means, that every node i hosts a value $x_i(t)$, a weight $w_i(t)$ and the objective is that all $\tilde{a}_i(t) := x_i(t)/w_i(t)$ converge towards $a := \sum_{i\in\mathcal{V}} x_i(0) / \sum_{i\in\mathcal{V}} w_i(0)$.

2.2 Fault Tolerance Aspects The decentralized computation of aggregate information in distributed systems is a fundamental operation which serves as building block for more complex operations. By their principles, gossip-based algorithms are very flexible and have a high fault tolerance potential. The crucial requirement for such gossip-based algorithms to converge to the correct result is that the (global) information is conserved during the whole course of the computation. Clearly, the hardness of this condition is determined by the concrete kind of operation to be performed. E. g., if we want to compute the minimum or the maximum of the data hosted by the nodes, the algorithm only has to ensure for all times $t > 0$ (using the notation of the previous section)

$$\min_{i\in\mathcal{V}} \tilde{a}_i(0) = \min_{i\in\mathcal{V}} \tilde{a}_i(t), \quad \max_{i\in\mathcal{V}} \tilde{a}_i(0) = \max_{i\in\mathcal{V}} \tilde{a}_i(t). \tag{2.1}$$

Such conditions are relatively easy to guarantee and are hardly effected by any kinds of failures since during the computation a high degree of natural redundancy is accomplished, e. g., more and more nodes hold the correct minimum/maximum. Moreover, one should note the close relationship between the computation of such discrete aggregates and the problem of spreading rumors (see e. g., [6]).

As stated in the previous section, we focus in this paper the computation of weighted averages. For this problem, the conservation property the algorithms have to ensure translates for all $t > 0$ to

$$\frac{\sum_{i\in\mathcal{V}} x_i(0)}{\sum_{i\in\mathcal{V}} w_i(0)} = \frac{\sum_{i\in\mathcal{V}} x_i(t)}{\sum_{i\in\mathcal{V}} w_i(t)}. \tag{2.2}$$

In contrast to the conservation properties (2.1) the conservation property (2.2) is basically affected by any kind of failure in the system. Thus, this property

is highly non-trivial to guarantee and explicit fault tolerance mechanisms are needed. For exactly this reason we consider this problem in this paper and investigate up to which extent fault tolerance at the algorithmic level can also be provided for this much harder problem.

While the search for efficient distributed aggregation algorithms has received a lot of attention (cf. [5,7]), non-trivial fault tolerance aspects were only addressed recently [1, 2, 4, 8]. Fault tolerance at the aggregation level is inherently challenging, since it requires the development of strong fault tolerance techniques solely working at the algorithmic level. Moreover, well-established fault-tolerance techniques like ABFT (cf. [9]), middleware-based approaches like (diskless) checkpointing (see, e. g., [10]) or purely hardware-based approaches like triple modular redundancy (cf. [11]) can not be efficiently utilized due to the simplicity and atomicity of a single aggregation.

Degrees of fault tolerance Fault tolerance is not clearly defined and different fields and applications require different notions of fault tolerance. To distinguish in the following between the fault tolerance abilities of the discussed methods we consider the following exemplary notions of fault tolerance ordered from weaker to stronger notions: an algorithm A tolerates a failure F if...

(H1) ...the computation does not crash in an uncontrolled way and information about the occurrence of the failure is available (after termination).

(H2) ...the computation terminates successfully, but the computed result is not necessarily correct.

(H3) ...the correct result is computed with a, possibly failure-dependent, a priori unknown overhead. Concretely, handling a failure may in some situations be less efficient than starting the computation from scratch.

(H4) ...the correct result is computed with a bounded, reasonably small, overhead. By reasonably small we mean that the process of recovering from a failure takes (much) less effort than restarting the computation.

(H5) ...the correct result is computed without overhead.

The nature of gossip-based algorithms guarantees fault tolerance at least in the sense of (H2). While the requirements of (H5) are out of reach for the averaging problems considered (in contrast to discrete aggregates where this superior notion of fault tolerance is reasonable) the interesting question—which also the existing theory leaves open—is how well (certain kinds of) failures are tolerated in practice, i. e., if we have to pay an arbitrarily high price for a correct result (cf. (H3)), or if we can efficiently compute correct results despite failures occurring during the computation (cf. (H4)). Consequently, when we henceforth state that an algorithm handles a certain kind of failure, we mean that it handles the failure in the sense of (H4).

We review in the following the most promising fault tolerant aggregation approaches which aim to tackle the problem of fault tolerant distributed aggregation and evaluate their strength and (practical) limitations. The subsequent discussion of these algorithms can obviously only cover the fundamental properties and different approaches of the algorithms. For a detailed exposition we refer to the literature [1–5].

2.3 Algorithms Especially for the sake of performance comparisons we include the non fault tolerant *Push-Sum algorithm* (PS) [5] (see Figure 1) in our investigations. In the PS algorithm initially every node holds two values: the actual input data and a scalar weight which determines the type of aggregate which shall be computed (sum, average, etc.). In each iteration of the algorithm every node picks a random neighbor and sends half of its data to the selected neighbor. All received values are simply added to the respective local values. The central property for ensuring the fully accurate computation of the target aggregate proved to be the so-called global *mass conservation* (see, e. g., [5]), which states that the initial data has to be preserved during the whole course of the computation (cf. Equation 2.2). Obviously, any kind of system failure hurts this property and thus, an aggregation algorithm can only be fault tolerant if it can guarantee mass conservation by purely local means. The crucial property responsible for many good properties of the PS algorithm is the monotone (and exponentially fast) convergence with respect to the maximal local error $E_{\max}$ (cf. Section 3.2). Thus, the network-wide worst local approximate can only improve over time. As a baseline for the performance of distributed aggregation algorithms we consider the complexity of (optimal) deterministic parallel reduction algorithms (cf. [12]). For well connected networks, e. g., networks with a short diameter, the PS algorithm was shown to scale logarithmically with the number of nodes [5,7] exactly like optimal deterministic approaches.

LiMoSense (LMS). The basic idea behind *LiMoSense* [2] is to incorporate fault tolerance mechanisms into the PS algorithm while not changing the algorithms

1: Local variables: $u_i = (s_i, w_i) \in \mathbb{R}^2$ 2: Initial values: $u_i \leftarrow (x_i, 1)$ 3: **for all** received pairs r **do** 4: $u_i \leftarrow u_i + r$ 5: **end for** 6: $k \leftarrow$ choose a neighbor uniformly at random 7: $u_i \leftarrow u_i/2$ 8: Send u_i to k	1: Local variables: $u_i = (s_i, w_i),\ f_{i,j} = (f^s_{i,j}, f^w_{i,j}) \in \mathbb{R}^2$ 2: Initial values: $u_i \leftarrow (x_i, 1),\ f_{i,j} \leftarrow (0, 0)$ 3: **for all** received pairs $f_{j,i}$ **do** 4: $f_{i,j} \leftarrow -f_{j,i}$ 5: **end for** 6: $k \leftarrow$ choose a neighbor uniformly at random 7: $f_{i,k} \leftarrow f_{i,k} + (x_i - \sum_{j \in \mathcal{N}_i} f_{i,j})/2$ 8: Send $f_{i,k}$ to k

Figure 1: Left: Push-Sum [5] without any (explicit) fault tolerance mechanisms. Right: Push-Flow [3] which utilizes the graph theoretical flow concept to become fault tolerant. Push-Flow reduces to Push-Sum in the absence of failures.

principal shape and behavior in failure-free environments. This is done by keeping (locally at each node) for each communication link to a neighboring node *histories* (sums) of sent and received values. By transmitting in every iteration the complete history (instead of the current message only), a receiving node can, e. g., easily recover from lost messages. To keep these ever growing histories reasonably small, a push-pull mechanism is supposed to be applied periodically. In failure-free cases LiMoSense reduces to the PS algorithm and hence, fast convergence is guaranteed.

Flow-Updating (FU). In contrast to LiMoSense which extends the PS algorithm by fault tolerance mechanisms, *Flow-Updating* [1] is an entirely independent approach. It is based on the following two ideas: (*i*) a node computes its local estimate of the aggregate as average over the most recent estimates it got from its neighbors. (*ii*) The graph theoretical flow concept is used for communicating local information fault tolerantly, i. e., instead of keeping histories as in LiMoSense the nodes in Flow-Updating keep for each communication link a flow variable $f_{i,j}$ which describes the amount of data which is "flowing" from node i to j. In contrast to histories, flow variables stay small by definition and no push-pull mechanisms are required. Moreover, by ensuring flow conservation (i. e., $f_{i,j} = -f_{j,i}$) the global property of mass conservation translates to an entirely local property which is the origin of the extraordinary high fault tolerance potential provided by the flow concept.

Push-Flow (PF). In principal the *Push-Flow algorithm* [3] aims to combine the positive aspects of Flow-Updating and LiMoSense, i. e., utilizing the flow concept while providing an algorithm which is in the absence of failures equivalent to the PS algorithm. While in Flow-Updating the flow concept is used as one ingredient of an entirely new algorithm, it was recently shown, that the flow concept can even be used in a more general way for deriving fault tolerant variants of gossip-based algorithms without changing their shape and properties in failure-free environments. Thus, the PF algorithm can be viewed as the special outcome of this methodology applied on the PS algorithm (see Figure 1). Moreover, this kind of transformation can e. g., be done for general classes of gossip-based aggregation algorithms as described in [7] or average consensus algorithms [13]. Consequently, all of the derived algorithms have the same fault tolerance properties and therefore, we only consider the PF algorithm in our evaluations.

Push-Cancel-Flow (PCF). The *Push-Cancel-Flow algorithm* [4] is a novel advancement of the PF algorithm which leads to significant qualitative improvements in terms of numerical accuracy and computational efficiency (when system failures occur). The theoretical advance of the PCF algorithm over the PF algorithm is, that the flow variables in the PCF algorithm converge to the target aggregate whereas they converge to arbitrary values in the case of the PF algorithm. This seemingly simple theoretical difference is responsible for all of the (practical) advantages of the PCF algorithm over the PF algorithm and other competing approaches (cf. Section 4).

In general there are two ways how the algorithms overcome system failures: first, certain types of failures, e. g., message loss, get corrected solely by executing the algorithm without any explicit failure detection or correction. More difficult (permanent) failures require in contrast to that knowledge about the kind and occurrence of the failure such that a failure handling

is possible. This is done by algorithmically excluding the failed component(s) from the computation. Concretely, by setting the respective history or flow variables to zero. Even though LiMoSense, the PF algorithm and the PCF algorithm produce results which are (theoretically) equivalent to the PS algorithm in failure-free environments, they can not preserve all properties of the PS algorithm when failures occur. Most notably, the monotone convergence with respect to the maximal local error $E_{\max}$ (cf. Section 3.2) gets lost in all fault tolerant algorithms.

While the four sketched fault tolerant aggregation algorithms provide superior fault tolerance properties compared to other approaches for distributed aggregation (see, e. g., [7, 13]), it is not clear up to which extent the theoretically expected fault tolerance can be observed in practice. This is due to the fact that the theoretical work on failure recovery usually only deals with the principal ability of tolerating system failures. Concretely, that means that it is (for many kinds of failures) assumed that the algorithm can be executed without any failures as long as it needs to overcome a previously occurred system failure. This unsatisfying circumstance is one of the motivating aspects of the present in-depth experimental study.

3 Experimental Methodology

In this section we discuss the formal background of our experimental work and define a concise methodology to objectively evaluate the algorithms discussed in Section 2.

3.1 System Failures of Interest

As a preliminary step, we classify and precisely define the system failures which are of interest in the context of robust distributed aggregation algorithms. Moreover, in Table 2 we depict which fault tolerance properties can be expected from each algorithm based on the existing theoretical results. As stated in Section 2, we expect an efficient failure handling of the algorithms and therefore a $\surd$ in Table 2 means that an algorithm handles the respective failure in the sense of (H4). Later, in Section 4, we update this comparison from a practical point of view (cf. Table 3) based on our experimental work and observe a substantial gap between theory and practice, i. e., we observe often only fault tolerance behavior in the sense of (H3).

3.1.1 Categorization of Failure Types

We categorize the system failures we are interested in with respect to three properties: the duration of the failure, the available information about the failure and the affected component. Moreover, we will use a systematic naming convention for providing a concise notation. Concretely, we will abbreviate the considered failures by `<duration><information>-<affected component>` where duration and information about the failure are specified by a single letter and the failed component by two letters, respectively. In the following we discuss the different characteristic properties of the considered system failures.

Duration. In terms of duration, we distinguish *transient* (t) and *permanent* (p) failures. In the case of a transient failure, a certain component of the system fails for a finite amount of time. After this time span, the system is operating as before and any effects which are due to this failure are corrected, e. g., if a communication link transiently fails, a successful communication is not possible for a finite amount of time, but after this time span, the link is operating as before. We speak about a permanent failure, when the effect of a system failure gets never corrected. It is obvious, that permanent failures are usually more difficult to handle than transient failures.

Information about failure. We distinguish three kinds of system failures: *unknown* (u), *known* (k) and *neat* (n) failures. In the case of an unknown failure, no information about the existence and impact of a system failure is available on the algorithmic level. In the case of a known failure, all neighbors of a node are assumed to be informed about the existence and type of failure immediately after the failure occurred. We speak about a neat failure, when a component knows about an upcoming problem and has enough time to successfully complete a failure handling routine, e. g., a sensor in a sensor network which is running out of battery can (neatly) sign off from a running computation by sending all its data to another node. Neat failures are easier to handle than known failures, and unknown failures are the most difficult to handle.

Affected component. In this work, we consider that the following four components may be affected by a failure during an aggregate computation: a *link* (li), an entire *node* (no), *algorithm data* (ad) and *initial data* (id). By a link we mean any kind of communication link and we speak about a *link failure* whenever a message from a node is not communicated (correctly) to the designated receiving node. As *node failure* we denote any kind of failures which result in a complete crash of a node, e. g., an operating system crash caused by a defective hardware component. We say that the algorithm data is affected by a failure, whenever one or more local variables of an algorithm get corrupted during the course of an aggregate computation, e. g., by a bit flip. Failures of the initial data are defined accordingly and denote any kind of corruption of the

Failure	PS	LMS	FU	PF	PCF
`tk-li`	√	√	√	√	√
`pk-li`	√	√	√	√	√
`tu-li`	—	√	√	√	√
`pu-li`	—	—	—	—	—
`pn-no`	√	√	√	√	√
`tk-no`	—	√	√	√	√
`pk-no`	—	√[1]	√[1]	√[1]	√[1]
`tu-no`	—	√	√	√	√
`pu-no`	—	—	—	—	—
`pu-ad`	—	—	√	√	√
`pu-id`	—	—	—	—	—

Table 2: Expected theoretical fault tolerance properties of Push-Sum [5], LiMoSense [2], Flow-Updating [1], Push-Flow [3] and Push-Cancel-Flow [4]. Cf. Table 3 for the empirically observed fault tolerance. Failures are abbreviated according to the definitions in Section 3.1.1.

[1]The initial data of the failed node is excluded from the aggregate.

data which is initially given to the nodes.

3.1.2 Failure description In principal the characteristic attributes we defined in Section 3.1.1 can be combined arbitrarily to define the set of system failures of interest. Not all of these combinations have meaningful interpretations and are of practical relevance and hence, we will only consider a certain subset of all possible combinations (cf. Table 1). In our simulations we will in principal (at certain points of time) inject single failures of a specific type to study their impact, i. e., we do not have to define a statistical failure model. The only exception is the failure `tu-li`. Here, we use $0 \leq p < 1$ as the *message loss rate* and the failures in the message transmissions are modeled as independent Bernoulli processes. Concretely, for any message which is sent in the simulator, we flip a coin to decide (with probability $1-p$) whether the message is successfully communicated or (with probability p) that `tu-li` occurs and the message gets lost.

Table 1 shows the selection of system failures we consider to be relevant in the context of gossip-based aggregation. Additionally, we give detailed failure description and practical examples of the described failures. Moreover, we depict in Table 2 the (theoretical) fault tolerance we can expect based on the existing literature. In Section 4 we evaluate up to which extent these expectations are fulfilled in practice and observe substantial gaps between theory and practice (cf. Table 3).

3.2 Objectives and Evaluation Procedure As discussed in Section 2, we consider in this paper the problem of scalar aggregation. Moreover, we saw that gossip-based aggregation algorithms are iterative processes which provide at each point of time an estimate for the target aggregate. To quantify the accuracy achieved at a certain point t in time, we use the (relative) maximal local error $E_{\max}$ which we define as $E_{\max}(t) := \max_{i=1}^{n} |\tilde{a}_i(t) - a|/|a|$ where a denotes the target aggregate and $\tilde{a}_i(t)$ denotes the local approximate of node i at time t. That means, if $E_{\max}(t) \leq \varepsilon$ holds, then all nodes have at least an ε-approximate of the target aggregate. Hence, whenever we specify a target precision τ, we say that an algorithm reached this target precision (at time t) when $E_{\max}(t) \leq \tau$. To gain more insights on how the network-wide approximates at a certain point of time look like, we also consider the minimal local error $E_{\min}(t) := \min_{i=1}^{n} |\tilde{a}_i(t) - a|/|a|$ and the mean local error $\mu_E(t) := \sum_{i=1}^{n} |\tilde{a}_i(t) - a|/(n \cdot |a|)$ with its corresponding standard deviation σ_E. In case we set in our simulations a target precision τ and an algorithm does not reach it, i. e., $E_{\max}(t) > \tau$ for all times t, we abort the computation after a predefined maximal number of iterations (usually 1000), and report the accuracy measures for the last completed iteration as achieved accuracy. While the usage of $E_{\max}$ as measure for the achieved accuracy guarantees that all nodes reached the prescribed target accuracy, existing experimental work (see, e. g., [1, 2]) often provides measures like the mean square error (MSE) [2] or the root mean square error (RMSE) [1] where no accuracy guarantees for a single node can be given. Such differences in the accuracy evaluation are fundamental and have a big influence on the computed results and their interpretation.

A central question in the experimental evaluation of gossip-based algorithms is how to account for the various sources of randomness in the experimental results. Concretely, we have to take into account three sources of randomness which strongly influence the behavior of the algorithms in our evaluations: (*i*) the communication schedules, (*ii*) the input data and (*iii*) the coin flips which are used for deciding whether or not a failure modeled by a Bernoulli process, e. g., `tu-li`, occurs. It is clear that varying any of these parameters will lead to different results, e. g., in terms of iterations needed for convergence. Thus, to account for these variations, we systematically vary all of these parameters and report statistical measures computed over a number of runs instead of presenting the result of a single run. A sufficiently large number of these repetitions guarantees meaningful unbiased results and we will also elaborate more details on how the number and kind of repetitions

Failure	Description	Practical example
`**-li`	A communication link fails, i. e., a successful transmission along this link is not possible.	
`tk-li`	All nodes connected by the failed link are informed at the begin and at the end of the finite failure period.	Transiently released wire
`pk-li`	All nodes connected by the failed link are informed about the permanent failure of the link after the link fails.	Permanently released wire
`tu-li`	Sent messages get silently dropped for a finite time period. The sender believes in the successful transmission of its message and neither sender or receiver are informed about the failure in the communication.	Message loss during communication
`pu-li`	The entire communication over the link gets silently dropped. Neither of the nodes connected via this link has information about the failure.	Permanently broken wire
`**-no`	A node entirely fails, i. e., no other node can communicate with the failed node and/or access its local data.	
`pn-no`	Before it permanently fails from a certain point of time on, the node can launch and complete a failure handling routine.	Sensor running out of battery
`tk-no`	All neighboring nodes are informed at the beginning and at the end of the finite failure period.	Transiently shut down node
`pk-no`	Before the node permanently failed, all neighboring nodes are informed about the failure.	Permanently shut down node
`tu-no`	All messages intended for the failed node get dropped for a finite period of time Neither of the neighboring nodes has information about the failure.	Unexpected system reboot
`pu-no`	All messages intended for the failed node get dropped from a certain point of time on. Neither of the neighboring nodes has information about the failure.	Crash of node
`pu-ad`	Local data (variables) maintained by the algorithm gets silently corrupted.	Bit-flip(s) in local variables of an algorithm
`pu-id`	The initial data stored by the algorithm gets silently corrupted.	Bit-flip(s) in the (variables storing) initial data

Table 1: Overview of system failures considered with failure description and practical examples

influences the achieved results in our experimental work in Section 4.

One of our central aims is to provide a comprehensive and reproducible study on how currently known theoretical properties of fault tolerant gossip-based aggregation algorithms translate into practice. Targeting the aim of reproducibility we designed our simulator in such a way that any kind of needed input data is read from permanently stored files. That means, to perform a simulation run with our simulator, we have to provide (*i*) the input data for the nodes which has to be aggregated, (*ii*) the communication schedule for every node (i. e., the neighbor which is called from node i in round t) and (*iii*) the coin flips which are necessary to decide whether certain failures occur or not. All of this data is provided via separate input files and we compiled an extensive test data set for our experimental evaluations which is also available online [8] (cf. Section 4). Furthermore, whenever we compare two or more algorithms in our experimental work, these algorithms use exactly the same parameters, i. e., the (random) communication schedules, (random) input data and coin flips are exactly the same and hence, the possible difference in the result is only due to the properties of the respective algorithms. Providing the test data set we compiled [8] is a central aspect since experimental results in the existing literature are hard to reproduce due to the lack of concrete environmental data (communication schedules, input data, etc.). Moreover, we plan to make our whole simulation environment publicly available after

completing our investigations.

3.3 Simulation Environment The central goal of our present experimental work is to study up to which extent the theoretically predicted properties of the considered algorithms hold in practice. That especially means, that we have to run our evaluations in an environment according to the formal results available in the literature. Concretely, for all of the considered algorithms, that means that we have to ensure an execution of the algorithms (at the nodes) in synchronous rounds, i. e., in a round every node executes one iteration of the present algorithm. This guarantees that we can gain insights from our experimental work with respect to existing theoretical results. We carry out our simulations with a self-made simulator which allows us to vary all parameters of interest in the considered algorithms. Despite our simulator runs sequentially, the results are computed as if the nodes would run the algorithms fully parallel in synchronous rounds. Therefore, we obtain exactly the same results as if we would run the experiments physically in parallel. How the considered algorithms actually behave in asynchronous environments which are out of scope of existing formal analyses is subject of ongoing research (cf. Section 5).

Since we aim for a clear comparison of the considered algorithms and for all of those the single node degrees of the nodes play an important role, we consider in the present study only topologies with a regular connection graph. Due to the regularity, i. e., every node has the same number of neighbors, we avoid any positive or negative outliers due to the use of a certain topology. On the other hand, we use different kinds of regular topologies to also take the influence of the topology on the performance under consideration. Concretely, our simulator can simulate any kind of k-ary d-cubes [14] as network connection graph. Especially, we will consider 2/3D grids and tori as well as hypercubes due to their neat formal properties, i. e., the considered algorithms benefit from a small network diameter (in terms of fast convergence) and from a small node degree (in terms of fault tolerance) which are both $\log_2(n)$ for a hypercube. Thus, hypercubes are a prime example for a topology where gossip-based aggregation works extremely well and where we can observe in practice (cf. Section 4) the theoretically predicted scaling of $\mathcal{O}(\log n + \log \varepsilon^{-1})$ [7] for a network of n nodes and a target precision (in terms of $E_{\max}$) of ε. This scaling behavior corresponds to the optimal scaling of deterministic aggregation algorithms (cf. [12]).

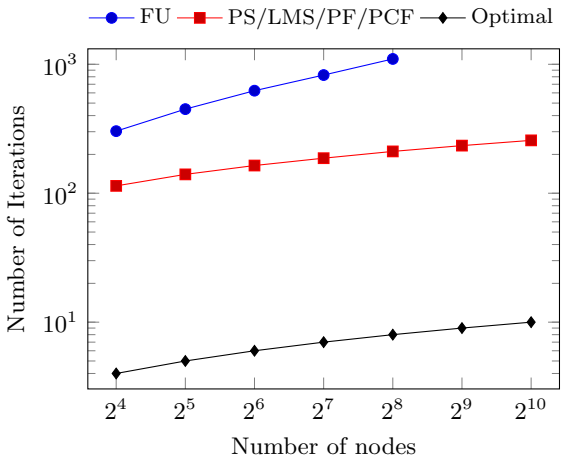

Figure 2: Number of iterations required to converge for Flow-Updating and all Push-Sum-based approaches. Simulation environment: hypercube topology, no message loss, convergence criterion $E_{\max} \leq 10^{-13}$ (or 1000 iterations), communication schedules `com-hc-2t8.`$\{1 \dots 25\}$, input data `inp-us-2t8.`$\{1 \dots 25\}$. See also Table 5.

4 Evaluation of Algorithms

In the following, we briefly discuss the evaluation data set [8] we compiled for our experimental studies and sketch our experimental results.

4.1 Evaluation Data Set As discussed in Section 3, our simulator reads any kind of environmental data from permanently stored files. For the sake of reproducibility, we compiled an extensive evaluation data set [8] for gossip-based aggregation algorithms consisting of a collection of (random) communication schedules for different topologies, different (random) input data sets as well as outcomes of coin flips required for the decision if `tu-li` occurs or not, i. e., if a message gets silently dropped or not. All communication schedules are drawn from an appropriate discrete uniform distribution depending on the used topology. Concretely, we provide communication schedules for fully connected networks (`com-fc-`), hypercubes (`com-hc-`), 2D grids (`com-2d-`) as well as 3D tori (`com-3d-`) in each case for $n = 2^d$ with $d = 4 \dots 10$. Moreover, for each network size we provide corresponding input files containing the needed coin flips (`ber-`). As inputs we consider standard uniformly (i. e., on $[0, 1]$) distributed values (`inp-us-`) as well as standard exponentially (i. e., $\lambda = 1$) distributed values.

In this extended abstract we provide exemplary insights into the results we obtained in our experimental studies. The detailed presentation of additional exper-

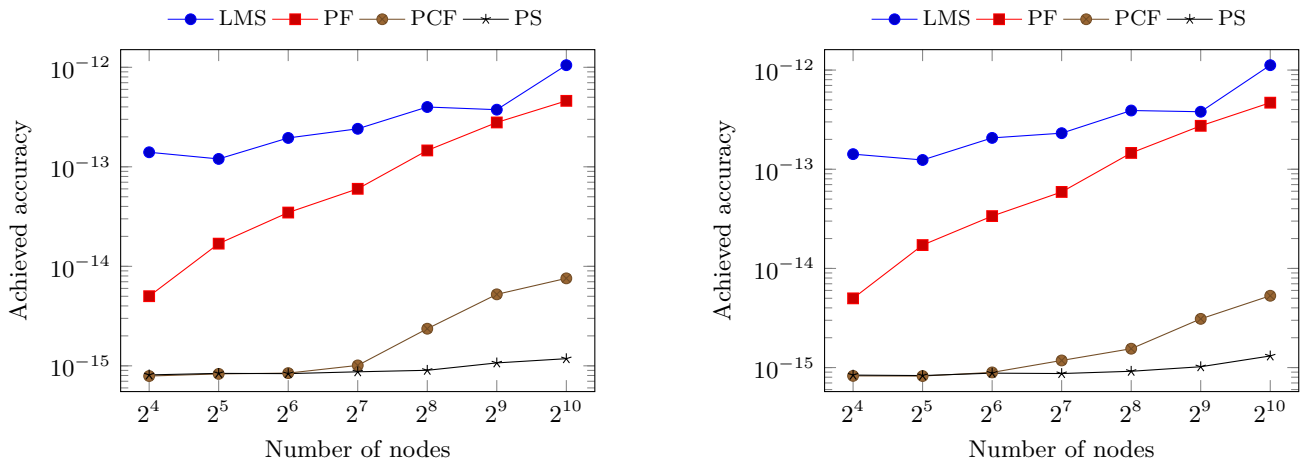

Figure 3: Achieved accuracy for increasing number of nodes. Simulation environment: 8D hypercube topology, no message loss, convergence criterion $E_{\max} \leq 10^{-13}$ (or 2000 iterations), communication schedules `com-hc-2t8.`$\{1 \ldots 25\}$, input data `inp-us-2t8.`$\{1 \ldots 25\}$ (left) and `inp-es-2t8.`$\{1 \ldots 25\}$ (right). See also Tables 6 and 7.

iments is deferred to the full version of this paper. If not explicitly stated else, we will in this extended abstract always consider an 8D-hypercube, i. e., a network with 256 nodes and input data drawn from a standard uniform distribution.

4.2 Failure-free Scenarios Before we study the practical impact of system failures in Section 4.3 we assume in this Section first failure-free environment to discuss basic properties of the considered algorithms.

The number of repetitions. To account for the various sources of randomness in our simulations we always have to repeat a certain scenario (with input data of the same kind) sufficiently often such that a clear picture about the difficulty can be provided. In Table 4 we show how the variation of communication schedules and input data leads to a wide range of observed results.

Convergence speed. We investigate algorithms which compute global aggregates locally at each node. Optimal parallel algorithms need $\mathcal{O}(\log n)$ (synchronous) iterations (see e. g., [12]) to perform this computation but require a lot of assumptions on the environment which are usually hurt in the context of distributed computing. Moreover, from theory, we can expect that the PS algorithm (and all its fault tolerant variants) converge asymptotically equally fast on well-connected networks, i. e., in $\mathcal{O}(\log n + \log \epsilon^{-1})$ time. In Figure 2 we can observe this theoretical claim for the case of a hypercube network. Obviously, Flow-Updating shows an uncompetitive convergence speed which results from the averaging technique used by the algorithm (cf. Section 2). Since Flow-Updating shares all its fault tolerant properties with the PF algorithm we will henceforth only consider the more efficient PF algorithm.

Numerical accuracy. The existing theory about gossip-based aggregation ignores the fact, that we have to deal with floating point arithmetic on real world systems. Hence, it is not clear if the theoretical predictions on convergence, etc., can also be observed in practice. While non fault tolerant algorithms like the PS algorithm have no problem in achieving full precision, we generally observe that the discussed fault tolerant mechanism introduce numerical difficulties. In Figure 3 we can see that the achievable accuracy decreases unexpectedly fast with the number of nodes. Among the fault tolerant algorithms the PCF algorithm shows by far the best results.

4.3 Fault Tolerance Capabilities In the following we inject different kinds of failures during the computation and study their impact. Since all the considered fault tolerance mechanisms strongly dependend on the individual node degrees we avoid extreme outliers by considering regular network topologies. This assumption is not restrictive since the results clearly translate also to practically relevant (almost regular) topologies like random geometric graphs.

Silent communication failures. One of the most important failure scenarios in loosely coupled distributed environments are silently occurring communication failures, which we refer to by `tu-li` in our classification scheme. While existing work clearly shows that lost messages can be handled in principal, no results of the concrete costs of tolerating a certain rate of message

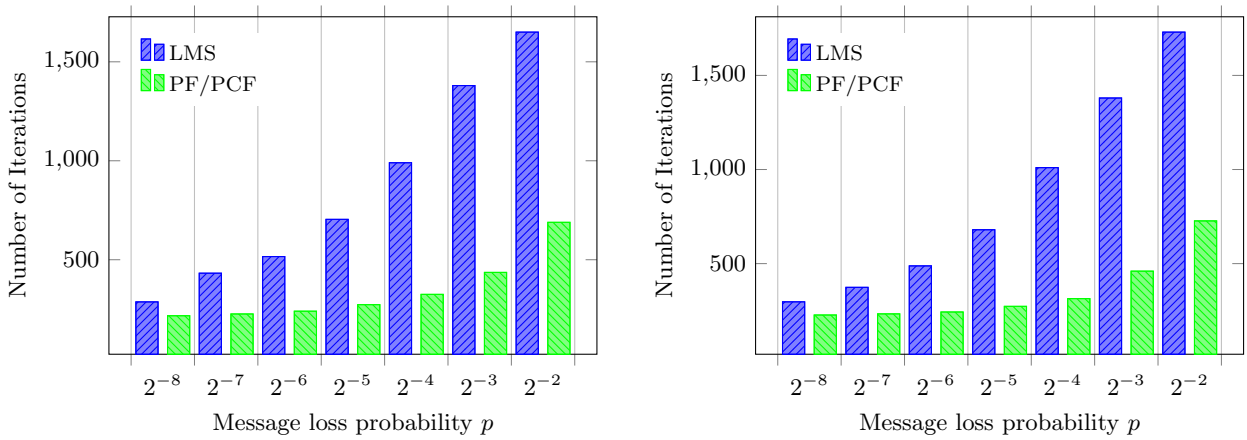

Figure 4: Impact of message loss on the convergence of LiMoSense and the PF/PCF algorithm. Simulation environment: 8D hypercube topology, message loss probability p, convergence criterion $E_{\max} \leq 10^{-13}$ (or 2000 iterations). Top: communication schedules `com-hc-2t8.`$\{1 \dots 25\}$, input data `inp-us-2t8.`$\{1 \dots 25\}$, coin flips: `ber-2t8.1`. Bottom: communication schedules `com-hc-2t8.1`, input data `inp-us-2t8.`$\{1 \dots 25\}$, coin flips: `inp-us-2t8.`$\{1 \dots 25\}$. See also Table 8.

loss are known.

As mentioned earlier, our simulator independently decides (based on coin flips given as input) for every communication if the communication (silently) fails or not. In Figure 4 we study the influence of an increasing message loss rate on the overall convergence. LiMoSense as well as the PF/PCF algorithm overcome this type of failure without any explicit failure handling. While in LiMoSense a failed communication can only be corrected by the successful transmission of a message of the same kind, i. e., a message with the same sender and receiver, all flow-based approaches allow for a bidirectional recovery. That means that any successfully transmitted messages between the two node can be used for the recovery. This theoretical advance explains the superior convergence speed of flow-based approaches which we empirically observe in Figure 4.

While the different approaches to handle message loss clearly differ in their efficiency, we observe in principle satisfactory results, since the overhead increases gracefully with the message loss rate. Summarizing, despite there are no formal results on the concrete overhead of tolerating a certain failure rate available, we get satisfying results in practice.

Permanent failures. In contrast to the handling of failures like `tu-li` which was shown to be possible without explicitly detecting or correcting occurring failures, tolerating permanent failures like `pk-li` requires a more explicit failure handling. In current algorithms this is done by algorithmically excluding the failed component from the aggregate computation, e. g., by setting flow or history variables to zero. Thus, such a failure handling is an "artificial" operation which does not follow the usual steps of the algorithm. Not surprisingly, such an intrusion into the algorithm has severe consequence and we basically observe a fall-back of the convergence to an arbitrary early stage independent from the accuracy which was already achieved (see Fig. 4 and Fig. 7 in [4] for graphical illustrations of this problem). This behavior can be studied Figure 5 (for LiMoSense and the PF algorithm) where we see that the overhead steadily increases the later the failure occurs. Contrary to that, the PCF algorithm shows only a minor constant overhead. For the cases of `pn-no` and `pk-no` we observe exactly the same problems since a failing node can in principal be interpreted as the failure of all its communication links.

The observation used in the PCF algorithm to gain its superior behavior for handling permanent failures is the following: in the PF algorithm the current estimate of a node i is computed as

$$\frac{x_i(t)}{w_i(t)} = \frac{x_i(0) - \sum_{j \in \mathcal{N}_i} f^x_{i,j}(t)}{w_i(0) - \sum_{\jmath \in \mathcal{N}_i} f^w_{i,j}(t)}.$$

If we now set $f_{i,k}$, $k \in \mathcal{N}_i$ to zero (due to a permanent failure) the fraction $x_i(t)/w_i(t)$ can change basically arbitrarily within the range of the initial data and thus, we will in general observe a *fall-back* in the convergence to a very early stage. Roughly speaking, the principal idea of the PCF algorithm is to achieve that $f^x_{i,k}(t)/f^w_{i,k}(t) \approx x_i(t)/w_i(t)$ for all $k \in \mathcal{N}_i$ because

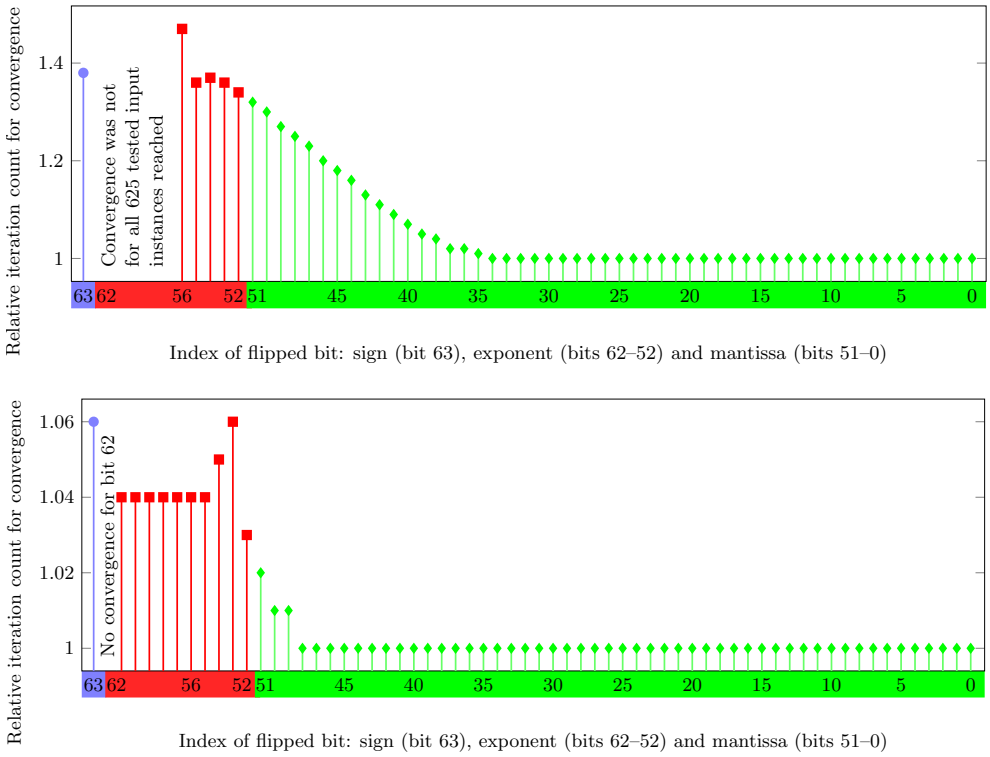

Figure 6: A single bit b gets flipped in the flow variable of a fixed node i after its local error was $\leq 10^{-8}$ (top) and $\leq 10^{-4}$ (bottom) for the first time, respectively. A strong relationship between the iterations required for convergence and the index of the flipped bit can be observed. Simulation environment: 8D hypercube topology, no message loss, convergence criterion $E_{\max} \leq 10^{-13}$, communication schedules `com-hc-2t8.`$\{1 \dots 25\}$, input data `inp-us-2t8.`$\{1 \dots 25\}$. See also Table 10.

this also guarantees that

$$\frac{x_i(0) - \sum_{j \in \mathcal{N}_i} f^x_{i,j}(t)}{w_i(0) - \sum_{j \in \mathcal{N}_i} f^w_{i,j}(t)} \approx \frac{x_i(0) - \sum_{j \in \mathcal{N}_i, j \neq k} f^x_{i,j}(t)}{w_i(0) - \sum_{j \in \mathcal{N}_i, j \neq k} f^w_{i,j}(t)}.$$

For details how this principal idea is realized concretely we refer to [4].

Silent data corruption. The shape of flow-based aggregation algorithms (in contrast to history-based approaches) allows in principal also the recovery from silently occurring data corruptions as long as the original input data is not affected. That means, flow-based approaches can in principle tolerate `pu-ad` but not `pu-id`. In Figure 6 we see for the case of the PF algorithm how the position of a single injected bit flip as well as the point of time when the failure was injected influences the convergence speed. Concretely, we observe on the one hand that many bit flips can be handled without any overhead whereas certain bit flips lead to high overheads and even convergence problems. The positions where no overhead occurs can be explained by the fact that in the PF algorithm (without failures) $E_{\max}$ is monotonously decreasing, i. e., we have at each point of time a certain range of values. Consequently, whenever a bit flip does not effect this range, the algorithm naturally overcomes failure. Clearly, the later a bit flip happens, the smaller $E_{\max}$ gets and thus, the more bits potentially lead to an overhead (cf. Figure 6). Moreover, this argumentation also shows, that not the amount of bit flips but the precise location and time of occurrence determines the resulting overhead.

To understand better why certain bit flips even preclude the computation of correct results we briefly recapitulate how real numbers are represented as (double precision) floating point numbers. A real number x in double precision is usually represented as

$$x = (-1)^s \cdot (1 + m \cdot 2^{-52}) \cdot 2^{e-1023}$$

with the *sign* $s \in \{0, 1\}$, the *mantissa* $0 <= m < 2^{52}$ and the *exponent* $0 < e < 2^{11} - 1$. Obviously, flips in the most significant bits of the exponent can have tremendous impact, especially if the exponent is around the bias value 1023.

At a first glance it seems to be irritating, that in Figure 6 we observe for the earlier occurring bit

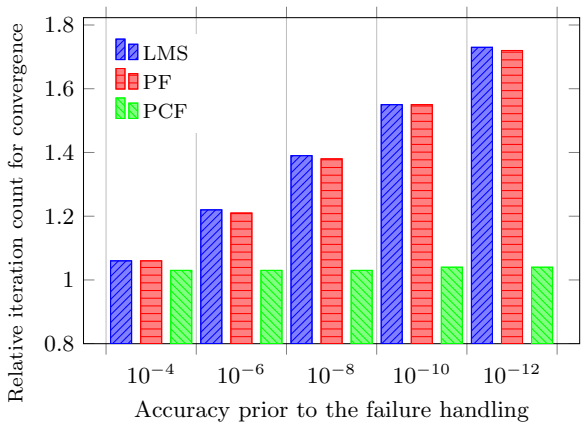

Figure 5: A single occurrence of `pk-li` is injected for a fixed link between nodes i and j after node i's local error was $\leq 10^{-x}$ for the first time. Except of the PCF algorithm we observe the more overhead the later the failure occurs which can be directly interpreted as improper failure handling. Simulation environment: 8D hypercube topology, no message loss, convergence criterion $E_{\max} \leq 10^{-13}$, communication schedules `com-hc-2t8.`$\{1 \ldots 25\}$, input data `inp-us-2t8.`$\{1 \ldots 25\}$. See also Table 9.

Failure	PS	LMS	FU	PF	PCF
`tk-li`	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
`pk-li`	$\surd$	$\sim$	$\sim$	$\sim$	$\surd^{2}$
`tu-li`	—	$\surd^{1}$	$\surd$	$\surd$	$\surd$
`pu-li`	—	—	—	—	—
`pn-no`	$\surd$	$\sim$	$\sim$	$\sim$	$\surd^{2}$
`tk-no`	—	$\surd$	$\surd$	$\surd$	$\surd$
`pk-no`	—	$\sim^{3}$	$\sim^{3}$	$\sim^{3}$	$\surd^{2,3}$
`tu-no`	—	$\surd$	$\surd$	$\surd$	$\surd$
`pu-no`	—	—	—	—	—
`pu-ad`	—	—	$\sim$	$\sim$	$\sim$
`pu-id`	—	—	—	—	—

Table 3: Practically observed fault tolerance properties of Push-Sum [5], LiMoSense [2], Flow-Updating [1], Push-Flow [3] and Push-Cancel-Flow [4]. Cf. Table 2 for the theoretically expected fault tolerance properties. Failures are abbreviated according to the definitions in Section 3.1.1.

[1] Histories are less efficient than flows, but the observed results correspond to the expectations from theory.

[2] The failure handling in the PCF algorithm can not preserve the monotone convergence of the PS algorithm, but shows only very small overheads.

[3] The initial data of the failed node is excluded from the aggregate.

flip only problems with the flip of the most significant exponent bit whereas the later occurring bit flip leads to problems with several of the most significant bits. This observation can be explained by the nature of the PF algorithm where we know that the flows gently develop (e. g. grow) over time to reach a stable equilibrium. In the case of the early bit flip, we always observe an exponent smaller than the bias value of 1023 and thus, only the change of the most significant bit leads to a (in absolute values) huge change of the variable. On the other hand, for the case of the later bit flip, the exponent sometimes exceeds the bias and therefore also the flip of the next few bits has a big impact in certain test cases.

Summarizing, we see a very diverse behavior in the handling of bit flips (depending on several parameters) which ranges from no overhead to a convergence wrong results. Hence, a fully reliable protection against `pu-ad` can not be provided by state-of-the-art methods.

In Table 3 we summarize the insights we gained by our experimental work. The purely theoretical fault tolerance properties for which we observed substantial problems in their practical behavior in our evaluations are marked by $\sim$ instead of $\surd$. A central issue which is revealed by Table 3 is that not all of the fault tolerance properties of the—in principal non fault tolerant—PS algorithm can be preserved in approaches striving for higher degrees of fault tolerance. Besides that, qualitative properties like full numerical accuracy get lost. While the PCF algorithm addresses already some of the issues discovered in this study and provides (major) qualitative improvements, it still shows several commonly observed weaknesses due to the similar techniques which are used. Hence, entirely new fault tolerance techniques seem to be required to further improve fault tolerance.

5 Conclusions

In theory, state-of-the-art robust gossip-based aggregation algorithms perform as well as their non fault tolerant counterparts but at the same time they provide high degrees of fault tolerance. Since the existing theoretical work does not address the performance impact of system failures, e. g., in terms of convergence speed, the theoretical results in the literature are not sufficient for a comprehensive evaluation and comparison of existing algorithms. We experimentally evaluated the theoretically most promising gossip-based fault tolerant aggregation algorithms in realistic (failure-prone) environments in order to understand how well the existing mechanisms for achieving fault tolerance work in practice.

In this experimental work we showed how simulation parameters like the (random) communication

schedules or the input data can strongly affect the achieved results and that a sound evaluation methodology is needed. We also illustrated that the fault tolerance mechanisms incorporated in the considered algorithms lead to unexpected practical problems such as numerical inaccuracies. Furthermore, we saw that several strategies for failure handling which work in theory are not competitive in practice due to arbitrary fall-backs in the convergence of the aggregate computation. Hence, the results presented in this paper provide deep insights into the behavior in practice and problems of state-of-the-art fault tolerant aggregation algorithms and reveal a big gap between the existing theory and the behavior in actual computations.

Outlook. According to the experimental observations we presented in this work, the novel PCF algorithm shows substantial improvements and superior fault tolerance properties over previously existing approaches. Nevertheless, some major failure types can still not be handled satisfactorily and not all (beneficial) properties of the baseline algorithms (e. g., monotone convergence in terms of $E_{\max}$) can be preserved. Thus, substantial theoretical investigations are required.

While we focused in this paper on scalar aggregation algorithms, our work naturally extends to more complex operations built on top of these atomic building blocks. In particular, in aggregation-based distributed matrix operations (see [15,16]), complex phenomena have to be expected due to the complexity of the algorithms and the potentially large number of distributed aggregation processes involved. Moreover, we work on a fine-grained ns-3 [17]-based simulation environment which allows for simulating aspects which are out of reach of the current theoretical results available for the discussed algorithms.

Acknowledgements

The research was funded by the Austrian Science Fund (FWF) under project number S10608.

References

[1] P. Jesus, C. Baquero, and P. S. Almeida, "Fault-Tolerant Aggregation by Flow Updating," in *Proceedings of the 9th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, ser. DAIS '09, 2009, pp. 73–86.

[2] I. Eyal, I. Keidar, and R. Rom, "LiMoSense – Live Monitoring in Dynamic Sensor Networks," in *7th Int'l Symp. on Algorithms for Sensor Systems, Wireless Ad Hoc Networks and Autonomous Mobile Entities (ALGOSENSORS'11)*, 2011.

[3] W. N. Gansterer, G. Niederbrucker, H. Straková, and S. Schulze Grotthoff, "Robust Distributed Orthogonalization Based on Randomized Aggregation," in *Proceedings of the Second Workshop on Scalable Algorithms for Large-Scale Systems*, ser. ScalA '11, 2011, pp. 7–10.

[4] G. Niederbrucker, H. Straková, and W. N. Gansterer, "Improving Fault Tolerance and Accuracy of Distributed Reduction Algorithms," in *Proceedings of the Third Workshop on Scalable Algorithms for Large-Scale Systems*, ser. ScalA '12, 2012, (*to appear*).

[5] D. Kempe, A. Dobra, and J. Gehrke, "Gossip-Based Computation of Aggregate Information," in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*, 2003, pp. 482–491.

[6] R. M. Karp, C. Schindelhauer, S. Shenker, and B. Vöcking, "Randomized rumor spreading," in *41st Annual Symposium on Foundations of Computer Science, FOCS 2000*, 2000, pp. 565–574.

[7] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah, "Randomized Gossip Algorithms," *IEEE/ACM Trans. Netw.*, vol. 14, pp. 2508–2530, June 2006.

[8] G. Niederbrucker and W. N. Gansterer, "An Evaluation Data Set for Gossip-based Aggregation Algorithms," September 2012. [Online]. Available: http://homepage.univie.ac.at/gerhard.niederbrucker/gossipData.zip

[9] K.-H. Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, Jun 1984.

[10] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *IEEE Trans. Parallel and Dist. Syst.*, vol. 9, no. 10, pp. 972–986, 1998.

[11] J. von Neumann, "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, pp. 43–98, 1956.

[12] R. Thakur and W. Gropp, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer Verlag, 2003, pp. 257–267.

[13] R. Olfati-Saber, J. A. Fax, and R. M. Murray, "Consensus and Cooperation in Networked Multi-Agent Systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 215–233, January 2007.

[14] W. J. Dally, "Performance analysis of k-ary n-cube interconnection networks," *IEEE Trans. Comput.*, vol. 39, pp. 775–785, June 1990.

[15] H. Straková, W. N. Gansterer, and T. Zemen, "Distributed QR Factorization Based on Randomized Algorithms," in *PPAM (1)*, ser. Lecture Notes in Computer Science, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., vol. 7203, 2011, pp. 235–244.

[16] H. Straková and W. N. Gansterer, "A Distributed Eigensolver for Loosely Coupled Networks," in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, ser. PDP 2013, 2013, (*to appear*).

[17] "The ns-3 network simulator." [Online]. Available: http://www.nsnam.org/

Conf.	Number of Iterations	$E_{\max}$	μ_E	σ_E^2
1/10	2.08e+02 ± 3.57e+00	8.75e-13 ± 1.01e-13	2.65e-13 ± 4.92e-14	3.53e-26 ± 1.03e-26
1/50	2.08e+02 ± 3.92e+00	8.58e-13 ± 1.23e-13	2.44e-13 ± 4.65e-14	3.24e-26 ± 1.08e-26
1/100	2.08e+02 ± 3.99e+00	8.53e-13 ± 1.19e-13	2.42e-13 ± 4.53e-14	3.20e-26 ± 1.04e-26
10/1	2.02e+02 ± 3.33e+00	9.03e-13 ± 9.00e-14	2.72e-13 ± 4.27e-14	3.70e-26 ± 9.86e-27
10/10	2.07e+02 ± 4.71e+00	8.59e-13 ± 1.14e-13	2.51e-13 ± 4.56e-14	3.37e-26 ± 9.99e-27
10/50	2.07e+02 ± 4.49e+00	8.55e-13 ± 1.21e-13	2.44e-13 ± 4.67e-14	3.27e-26 ± 1.07e-26
10/100	2.07e+02 ± 4.46e+00	8.59e-13 ± 1.15e-13	2.46e-13 ± 4.44e-14	3.28e-26 ± 1.02e-26
50/1	2.05e+02 ± 3.61e+00	8.78e-13 ± 1.06e-13	2.54e-13 ± 3.94e-14	3.33e-26 ± 8.66e-27
50/10	2.08e+02 ± 4.57e+00	8.56e-13 ± 1.24e-13	2.46e-13 ± 4.69e-14	3.27e-26 ± 1.05e-26
50/50	2.07e+02 ± 4.39e+00	8.58e-13 ± 1.18e-13	2.45e-13 ± 4.58e-14	3.28e-26 ± 1.04e-26
50/100	2.07e+02 ± 4.38e+00	8.58e-13 ± 1.15e-13	2.47e-13 ± 4.38e-14	3.29e-26 ± 1.01e-26
100/1	2.05e+02 ± 3.23e+00	8.74e-13 ± 1.01e-13	2.56e-13 ± 3.99e-14	3.38e-26 ± 9.08e-27
100/10	2.08e+02 ± 4.58e+00	8.57e-13 ± 1.20e-13	2.46e-13 ± 4.56e-14	3.27e-26 ± 1.02e-26
100/50	2.07e+02 ± 4.39e+00	8.58e-13 ± 1.17e-13	2.45e-13 ± 4.54e-14	3.28e-26 ± 1.03e-26
100/100	2.07e+02 ± 4.36e+00	8.58e-13 ± 1.15e-13	2.47e-13 ± 4.35e-14	3.28e-26 ± 1.00e-26

Table 4: Simulation environment (for configuration i/j): 8D hypercube topology, no message loss, convergence criterion $E_{\max} \leq 10^{-13}$ (or 1000 iterations), communication schedules `com-hc-2t8.`{1 . . . `i`}, input data `inp-us-2t8.`{1 . . . `j`}.

	Uniformly distributed initial data		Exponentially distributed initial data	
n	FU	PS/LMS/PF/PCF	FU	PS/LMS/PF/PCF
2^4	1.12e+02 ± 6.31e+00	3.03e+02 ± 1.03e+01	1.14e+02 ± 6.53e+00	3.03e+02 ± 1.01e+01
2^5	1.39e+02 ± 6.42e+00	4.49e+02 ± 9.01e+00	1.40e+02 ± 6.40e+00	4.49e+02 ± 8.95e+00
2^6	1.61e+02 ± 5.36e+00	6.24e+02 ± 5.39e+00	1.64e+02 ± 5.44e+00	6.24e+02 ± 5.38e+00
2^7	1.84e+02 ± 4.28e+00	8.25e+02 ± 3.45e+00	1.87e+02 ± 4.47e+00	8.25e+02 ± 3.52e+00
2^8	2.06e+02 ± 4.29e+00		2.11e+02 ± 4.30e+00	
2^9	2.29e+02 ± 3.70e+00		2.34e+02 ± 3.62e+00	
2^{10}	2.52e+02 ± 3.14e+00		2.57e+02 ± 3.26e+00	

Table 5: Supplementary data for the experiments depicted in Figure 2

n	LMS	PF	PCF	PS
2^4	1.42e-13 ± 1.74e-13	4.99e-15± 7.56e-15	8.25e-16± 2.96e-16	8.40e-16 ±1.54e-16
2^5	1.24e-13 ± 1.15e-13	1.72e-14± 1.50e-14	8.22e-16 ±1.61e-16	8.31e-16 ±1.10e-16
2^6	2.07e-13 ± 3.42e-13	3.37e-14 ±5.63e-14	8.94e-16 ±3.25e-16	8.79e-16 ±1.03e-16
2^7	2.31e-13 ± 2.04e-13	5.90e-14 ±4.83e-14	1.18e-15± 1.46e-15	8.71e-16 ±1.14e-16
2^8	3.91e-13 ± 4.85e-13	1.46e-13 ±7.41e-13	1.55e-15± 1.61e-15	9.18e-16 ±1.56e-16
2^9	3.80e-13 ± 3.37e-13	2.74e-13± 7.09e-13	3.10e-15 ±1.14e-15	1.02e-15 ±2.50e-16
2^{10}	1.12e-12 ± 1.45e-12	4.69e-13 ±1.10e-13	5.30e-15 ±6.76e-15	1.31e-15 ±4.88e-16

Table 6: Supplementary data for the experiments depicted in the right part of Figure 3

n	LMS	PF	PCF	PS
2^4	1.40e-13± 1.69e-13	5.01e-15± 8.53e-15	7.90e-16± 1.48e-16	8.11e-16± 1.51e-16
2^5	1.20e-13± 1.13e-13	1.69e-14± 1.30e-14	8.30e-16± 1.63e-16	8.40e-16± 1.39e-16
2^6	1.95e-13± 2.92e-13	3.47e-14± 5.77e-14	8.45e-16± 4.39e-16	8.35e-16± 1.20e-16
2^7	2.41e-13± 2.37e-13	6.01e-14± 5.54e-14	1.01e-15± 6.64e-16	8.71e-16± 1.02e-16
2^8	3.99e-13± 5.34e-13	1.46e-13± 7.59e-13	2.36e-15± 1.17e-15	9.02e-16± 1.75e-16
2^9	3.75e-13± 3.22e-13	2.79e-13± 6.91e-13	5.23e-15± 1.01e-15	1.07e-15± 3.18e-16
2^{10}	1.05e-12± 1.33e-12	4.60e-13± 1.12e-13	7.57e-15± 4.31e-15	1.18e-15± 4.01e-16

Table 7: Supplementary data for the experiments depicted in the left part of Figure 3

	Fixed Input		Fixed Coin Flips	
p	LMS	PF/PCF	LMS	PF/PCF
2^{-8}	2.87e+02 ± 1.34e+02	2.17e+02 ± 2.71e+01	2.97e+02 ± 1.47e+02	2.27e+02 ± 7.71e+01
2^{-7}	4.32e+02 ± 4.07e+02	2.26e+02 ± 3.97e+01	3.74e+02 ± 2.12e+02	2.33e+02 ± 7.69e+01
2^{-6}	5.16e+02 ± 3.89e+02	2.40e+02 ± 4.48e+01	4.88e+02 ± 2.54e+02	2.43e+02 ± 7.64e+01
2^{-5}	7.04e+02 ± 3.87e+02	2.73e+02 ± 8.09e+01	6.80e+02 ± 3.17e+02	2.73e+02 ± 7.85e+01
2^{-4}	9.90e+02 ± 3.70e+02	3.25e+02 ± 9.97e+01	1.01e+03 ± 3.85e+02	3.14e+02 ± 7.67e+01
2^{-3}	1.38e+03 ± 3.11e+02	4.36e+02 ± 1.37e+02	1.38e+03 ± 3.58e+02	4.60e+02 ± 1.96e+02
2^{-2}	1.65e+03 ± 2.21e+02	6.89e+02 ± 1.82e+02	1.73e+03 ± 2.01e+02	7.27e+02 ± 2.10e+02

Table 8: Supplementary data for the experiments depicted in Figure 4.

	10^{-4}	10^{-8}	10^{-12}
LMS	2.19e+02 ± 1.13e+01	2.86e+02 ± 1.59e+01	3.57e+02 ± 1.57e+01
PF	2.19e+02 ± 1.13e+01	2.86e+02 ± 1.59e+01	3.57e+02 ± 1.57e+01
PCF	1.64e+02 ± 3.14e+00	1.64e+02 ± 3.37e+00	1.65e+02 ± 4.20e+00

Table 9: Supplementary data for the experiments depicted in Figure 5.

	Failure after 10^{-4}	Failure after 10^{-8}
Bit 63	2.19e+02 ± 1.10e+01	2.85e+02 ± 2.83e+01
Bit 56	2.15e+02 ± 9.53e+00	3.04e+02 ± 1.36e+02
Bit 55	2.15e+02 ± 9.55e+00	2.81e+02 ± 2.95e+01
Bit 54	2.17e+02 ± 1.01e+01	2.82e+02 ± 2.72e+01
Bit 53	2.18e+02 ± 1.04e+01	2.81e+02 ± 2.74e+01
Bit 52	2.13e+02 ± 8.88e+00	2.77e+02 ± 2.64e+01
Bit 51	2.11e+02 ± 7.35e+00	2.73e+02 ± 2.54e+01
Bit 50	2.09e+02 ± 5.99e+00	2.68e+02 ± 2.42e+01
Bit 45	2.06e+02 ± 4.30e+00	2.43e+02 ± 1.89e+01
Bit 40	2.06e+02 ± 4.30e+00	2.21e+02 ± 1.28e+01
Bit 30	2.06e+02 ± 4.31e+00	2.06e+02 ± 4.31e+00
Bit 20	2.06e+02 ± 4.30e+00	2.06e+02 ± 4.29e+00
Bit 10	2.06e+02 ± 4.31e+00	2.06e+02 ± 4.31e+00
Bit 0	2.06e+02 ± 4.29e+00	2.06e+02 ± 4.27e+00

Table 10: Supplementary data for the experiments depicted in Figure 6