SDLE 19/20: On the Push-Sum and Flow Updating Distributed Data Aggregation Algorithms in Asynchronous Settings

Alberto Faria (A79077), João Marques (A81826), Nuno Rei (A81918)

31 May 2020

1 INTRODUCTION

This document consists of the report pertaining to the practical assignment of the course entitled *Sistemas Distribuídos em Larga Escala*, part of the Integrated Master's Degree in Informatics Engineering at the University of Minho, lectured in the 2019/2020 school year. The assignment focuses on the analysis, simulation, and evaluation of distributed data aggregation algorithms, an essential component of modern distributed systems that enables the decentralized evaluation of important global properties [5].

We consider two gossip-based averaging aggregation algorithms: Push-Sum [6] and Flow Updating [3]. Our focus is on their behavior under an asynchronous system model. Since both algorithms were originally specified for synchronous systems, we begin by applying straightforward modifications that make them suitable for application in asynchronous settings.

In order to simulate the execution of these algorithms, a simple discrete-event simulator for networked systems was developed. Then, both algorithms were implemented in this simulator, which was used to conduct an evaluation of their performance under several network topologies and reliability conditions.

This report is structured as follows. Section 2 begins by defining the problem of distributed data aggregation. Section 3 then outlines the two algorithms contemplated in this work and their adaptations for asynchronous settings. The developed discrete-event simulator is described in Section 4, while Section 5 briefly discusses the implementation of both algorithms in this simulator. Section 6 presents and analyses the results of the conducted evaluation, comparing the performance of both algorithms under several scenarios, and Section 7 concludes the report with directions for future work.

2 DISTRIBUTED DATA AGGREGATION

In modern scalable distributed system designs, data aggregation plays an important role allowing the decentralized determination of meaningful global properties that can inform the execution of other applications [5]. Examples of such global properties include the network size, the average system load, and the total disk space available in a distributed storage system. Underlying these properties is system-wide the evaluation of functions such as `Count`, `Sum`, and `Average`.

In [5], the authors define the process of data aggregation as the computation of an *aggregation function*, which is any function f that takes as input a multiset of elements of a set I and produces as output an element of a set O (symbolically, $f : \mathbb{N}^I \to O$), emphasizing that the order in which elements are aggregated is immaterial and that duplicate elements may exist.

The authors further introduce the notion of a *decomposable* aggregation function. We avoid the precise definition here, but remark that `Count`, `Sum`, and `Average` are decomposable. Such functions may be decomposed into several computations involving sub-multisets of the multiset being aggregated, and are thus amenable to decentralized evaluation.

3 ALGORITHMS

A wide variety of algorithms for distributed data aggregation have been proposed, exhibiting different tradeoffs of accuracy, speed, and communication overhead. These algorithms may be organized into five main classes [5]: (i) hierarchy-based, (ii) sketch-based, (iii) averaging, (iv) randomized, and (v) digest-based. We briefly summarize the advantages and disadvantages of algorithms belonging to each of these classes, and refer the reader to [5] for information on their underlying mechanisms.

Hierarchy-based approaches rely on a specific underlying network topology. They are accurate and efficient, but do not tolerate failures. *Sketch-based* approaches in turn can be used in fault-prone scenarios but have limited accuracy. In contrast, *Averaging* algorithms exhibit significantly higher communication overhead but can tolerate high-failure rate scenarios (especially with regard to node churn) and are accurate, producing estimates that converge to the desired value over time. According to [5], *randomized* techniques are strictly inferior to other approaches, being slow, inaccurate, commonly restricted to the computation of specific aggregation functions (such as `Count`), and unreliable in the presence of faults. Finally, *digest-based* approaches support the computation of a wide range of aggregation functions but have limited accuracy and significant communication overhead.

The two algorithms contemplated in this work, Push-Sum [6] and Flow Updating [3], belong to the class of averaging algorithms. They employ a gossip protocol and are thus independent of the underlying network topology, producing estimates at every node which converge over time to the correct value (as long the network is connected). Both allow the computation of the `Average` function and decomposable functions derivable from it, such as `Sum` and `Count`.

We now detail the functioning of both the Push-Sum and Flow Updating algorithms. Since their original specifications assume a synchronous system model, and because we are interested in their behavior in asynchronous settings, we also describe simple modifications that make them suitable for use under asynchronous assumptions. We also assume the possibility of transient link failures, resulting in the undetectable loss of some messages.

3.1 Push-Sum

At each node i and time step or round $t \in \mathbb{N}$, the Push-Sum algorithm [6] maintains the following state: a sum $s_{t,i}$ and a weight $w_{t,i}$. Then, in each round t, each node i creates a message $(\frac{1}{2}s_{t,i}, \frac{1}{2}w_{t,i})$ and sends it to itself and to one of its neighbors, chosen uniformly at random. At the end of each round, each node i sets $s_{t+1,i}$ and $w_{t+1,i}$ to the sum of all sums and weights in messages received in that round. The estimate at a time step t is defined to be $s_{t,i}/w_{t,i}$.

By varying the values of $s_{0,i}$ and $w_{0,i}$, *i.e.*, the initial sum and weight at each node, one can compute several aggregation functions using this algorithm. To compute the `Average`, for instance, one sets the initial sum at each node to that node's value and all initial weights to 1. For `Sum`, the initial sum is set equivalently, but only one node sets its initial weight to 1, an all others set it to 0. Computing `Count` is similar, with each node setting its initial value to 1.

Relaxing the system model. Besides assuming a synchronous system model, the original specification of the Push-Sum algorithm also assumes that the loss of a message can be detected by the sender (in which case the specification dictates that the message be sent instead to the sender itself). This is because the algorithm produces incorrect results if any of the sent messages is lost, as for the estimates to converge to the correct global aggregate value, the invariant of *mass conservation* must be maintained [6].

To overcome this, we extend the protocol with a simple acknowledgment procedure, by which the receiver of a message must respond to the sender with a special acknowledgment message (which need not itself be acknowledged). If the original sender does not receive an acknowledgment message within a configurable amount of time, it retransmits the original message. This process repeats until the message is acknowledged. (Note that under this system model, the Push-Sum algorithm cannot tolerate the removal of links during execution, since otherwise a link could be removed after transmitting a message but before transmitting its acknowledgment. In this case, the original sender of the message would deadlock.)

Since under asynchronous assumptions the global notion of a round is invalid, we consider instead a local notion of round. Each node moves on to its next round as soon as the last message that it sent is acknowledged.

3.2 Flow Updating

The Flow Updating algorithm [3] takes inspiration from the graph theoretical concept of *flow*. Unlike Push-Sum, which work by iteratively exchanging "mass" between nodes, Flow Updating instead keeps initial inputs unchanged and exchanges and updates the flow at each link, leveraging the symmetry property of flows. The key idea is for each node to compute the current estimate from the initial input value and the current flow at each incident link.

Because of this, Flow Updating is natively tolerant of message loss, since messages merely update values instead of exchanging mass. Thus, the loss of a message may imply slower converge toward the correct aggregate value, but does not break safety.

The state maintained by each node is the following: (i) the node's original input value, (ii) the flow to each of the node's neighbors (initially zero), and (iii) an estimate for each of the node's neighbors (initially zero). The algorithm alternates between a message exchanging phase and a state updating phase. In the message exchanging phase, each node sends to each of its neighbors a message containing its local flow and estimate for that neighbor. In the state updating phase, at each node i, the following steps occur:

(1) For each neighbor j, if a message was received from that neighbor, the node updates its local flow and estimate for that neighbor, denoted by f_{ij} and e_{ij} with the *symmetric* value of the flow in that message (due to the symmetry of flows) and with the estimate in that message;
(2) The node updates its local estimate e_i with the value of $(v_i - \sum_j f_{ij} + \sum_j e_{ij})/(d_i + 1)$, where v_i is the input value at node i and d_i is the number of neighbors of node i;
(3) For all neighbors, $f_{ij} \leftarrow f_{ij} + (e_i - e_{ij})$ and $e_{ij} \leftarrow e_i$.

The procedure described up until here corresponds to the *multicast* version of the algorithm, in which nodes send messages to all neighbors in every round. In the *unicast* version, on the other hand, nodes send messages to only one neighbor in each round, and the last step only performs the updates for that neighbor.

Relaxing the system model. Like Push-Sum, the original specification of this algorithm assumes a synchronous system model. However, since Flow Updating tolerates message losses, no additional explicit retransmission or message loss detection mechanism is needed.

However, the algorithm does rely on the notion of a global round. To avoid this notion, we make the following adaptations. For the multicast version of the algorithm, a node moves on to the next local round after it has received a message from all neighbors. However, since messages may be lost, this may lead to stalling of a node if a link is temporarily broken but not quickly fixed. As such, we add a configurable "round timeout" parameter. After the given amount of time passes, the node unconditionally moves on to the next round event if not all expected messages were received.

In the unicast version of the algorithm, nodes do not receive messages from all neighbors in every round. In fact, there might be rounds in which they do not receive a message at all. However, the average number of messages received per node per round is 1. We thus apply the same rule as for the multicast version: as soon as a node receives a message, it moves on to the next round. If no messages are received, it moves on to the next round after the timeout for the current round expires.

4 SIMULATOR

Motivated by the study of the behavior of the algorithms presented above, we developed a simple discrete-event simulator capable of simulating both synchronous and asynchronous distributed systems. In this section, we outline the simulator's design and implementation.

4.1 Design

At a high level, the simulator takes the definition of an undirected graph representing a network, where vertices correspond to nodes and edges to communication links, and simulates the occurrence of certain events and the behavior of the nodes in response to these events.

Each node can react to three types of event: (i) simulation start, (ii) message reception, and (iii) notification reception. In response, each node may sent messages to one or more of its neighbors and/or schedule a local notification to be delivered after a configurable amount of time. (Nodes may also send messages to themselves, which are always reliably and immediately delivered.) The first kind of event occurs simultaneously for all nodes at the moment that the simulation starts. The second and third kinds are the direct consequence on actions taken by nodes.

The distribution of message transmission delays across each link is configurable, as is the percentage of messages that the link should drop. The simulator keeps track of a global "simulated" time variable, and each event (message or notification reception) is scheduled globally for occurrence at a specific time, depending on the latency of the communication link or delay configured for the notification. The user may also specify a global termination predicate, which determines when the simulation should terminate. The simulation stops unconditionally if no more events are scheduled to occur. Once the simulation has ended, the user can inspect several statistics such as the elapsed "simulated" time, the number of sent messages, and the number of received messages.

4.2 Implementation

We implemented the simulator using the Python 3.8 language. The code is split into three modules belonging to the `sim` package: (i) `sim.utilities`, (ii) `sim.types`, and (iii) `sim.simulation`.

The `sim.utilities` module contains miscellaneous utilities used by the other modules and of possible use to users of the simulator. The `sim.types` module defines the main types used by the simulator. These include the class `Node`, which is used to specify the behavior of a node, and the class `Link`, which determines the latency and reliability characteristics of a link. The `Network` class joins these pieces by representing a full networked systems, storing its nodes and links. This last class also includes several predefined functions for generating networks with certain common topologies, such as networks based on Euclidean distances and Barabási–Albert networks [1] (although the user is free to define network of any topology). Finally, the `sim.simulator` module exports function `simulate()` which can be used to run simulations on a given network.

As previously described, a simulation consists of iteratively triggering events at certain nodes and adding more events based on the nodes' behavior. The set of events to occur is maintained in a min-heap, ordered by the "simulated" time at which events are to occur. The `simulate()` function repeatedly pops an event from this heap and delegates its processing to the respective node, until either the heap is empty or the user-specified termination predicate is satisfied. The simulator uses fixed-precision arithmetic for time accounting to avoid precision loss during long-running simulations.

5 SIMULATING THE ALGORITHMS

To simulated the two algorithms contemplated in this work, Push-Sum and Flow Updating, they were implemented as in the simulator presented above. Their implementations are contained in modules `dda.pushsum` and `dda.flowupdating` of package `dda`, which depends on the `sim` package containing the implementation of the simulator.

The `PushSumNode` class specifies the behavior of a node running an instance of the Push-Sum algorithm, and is a simple translation into code of the specification given in Section 3.1. The behavior of nodes running the Flow Updating algorithm is in turn specified by the `FlowUpdatingNode` class. This class can be instantiated to run either the unicast or multicast version of the Flow Updating algorithm. Further, the implementation supports the concurrent aggregation of several functions (each exchanged message containing flow and estimate value updates for all functions), which is useful for the computation of aggregate functions derived from several averages, such as `Sum` = `Average` · `Count` where `Count` can be computed as the inverse of the average with one node with value 1 and all others with value 0.

The `dda.main` module provides a simple command line interface to run simulations of the implemented algorithms under several network topologies and link reliability configurations. (In the code provided alongside this report, the `sim/dda.main.sh` script may be used to easily run this interface.)

6 EVALUATION

Using the simulator presented above and the aforementioned Push-Sum and Flow Updating implementations, an evaluation of both algorithms was performed. We now describe the methodology used to conduct this evaluation, and then present and discuss the obtained results.

6.1 Methodology

The Push-Sum algorithm is configured with a retransmission timeout of 20 ms. For the Flow Updating algorithm, we evaluate both the unicast and the multicast versions, and configure them with a round timeout of 20 ms. The procedures are configures to compute the `Average`. The value at each node is drawn from a uniform distribution ranging from 0 to 1.

All algorithms are evaluated under networks of 1000 nodes and three topology types:

- Erdős–Rényi [2], node degree $d \approx \ln n$, connected;
- Watts–Strogatz [7], $p = 0.5$, node degree $d \approx \ln n$, connected;
- Barabási–Albert [1], $m = m_0 = 2$.

The message transmission latency in every link follows a Gamma distribution with mean $\mu =$ 10 ms and shape parameter $\alpha = 3$. We experiment with fully reliable links and links which drop 20% of the messages.

In each experiment, we periodically observe the elapsed "simulated" time, the number of sent and received messages, and a measure of the current estimation error. This measure is the Root-Mean-Square Error (RMSE), taking into account the estimates at every node and the true global average. For each evaluated scenario, experiments are run 30 times. Results presented next are the average of these runs.

6.2 Results

The obtained results are depicted in 1. One can see that all algorithms converge exponentially to the correct aggregate, as expected [6]. Further, for Push-Sum, results the introduction of dropped messages delays the convergence and increases the elapsed time and number of sent messages in around the same proportion. One can also observe that the Flow Updating algorithm converges, in general, faster, the only exception being the Erdős–Rényi topology with 20% message loss. In terms of convergence speed, the multicast version of Flow Updating is the clear winner, although it

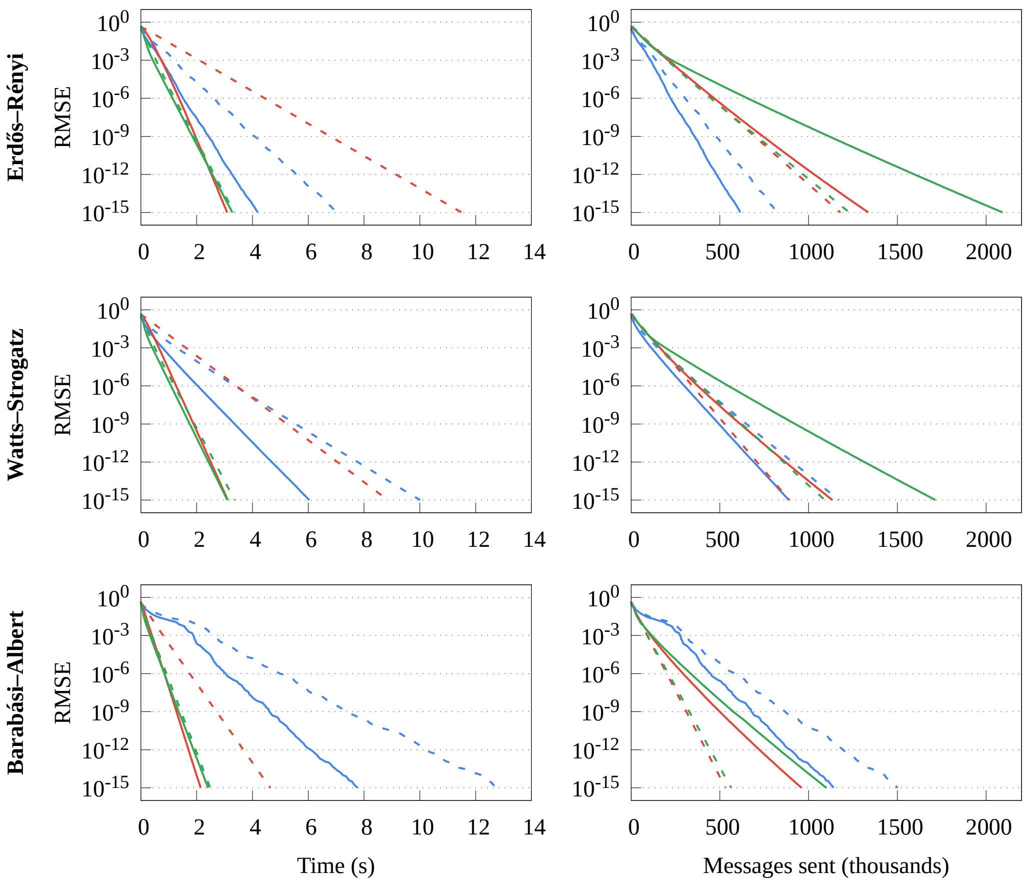

Figure 1. **Simulation results.** Root-mean-square error (RMSE) as a function of simulated time (left) and sent messages (right) for the Push-Sum (—), Unicast Flow Updating (—), and Multicast Flow Updating (—) algorithms with 0% (solid lines) and 20% (dashed lines) message loss on Erdős–Rényi, Watts–Strogatz, and Barabási–Albert topologies with 1000 nodes. Note the logarithmic scale.

does tend to send more messages that the unicast version. The multicast version also does not lose performance when subjected to message loss.

However, a strange phenomenon occurs with the Flow Updating algorithm regarding the number of sent messages. Specifically, when 20% of messages are lost, the algorithm ends up not needing to send as many messages to reach the same level of error as in the scenarios in which no messages are lost. Our hypothesis for this behavior is that, when messages are lost, nodes may have to wait for the round timeout to expire before continuing to perform work. This in turn reduces the rate of sent messages, but it appears that it does not have a proportionally large impact on the convergence speed. In order words, when no messages are lost, the algorithm may be sending more messages than would be needed to attain the same error in the same amount of time.

This is indicative of the impact on performance of the configuration values introduced by our adaptations to the algorithms. However, due to time constraints, we were unable to quantitatively evaluate the effect of the round timeout on the algorithm's performance, and instead leave this task for future work.

7 CONCLUSION

In this report, we began by defining the problem of distributed data aggregation and describing two averaging-based gossip algorithms for the computation of global aggregates in a distributed system. We then presented a simple discrete-event simulator targeted at the simulation of distributed systems. We implemented the two algorithms in this simulator and extended them with some simple adaptations to be applicable to an asynchronous system model with transient, undetectable link failures. Finally, an evaluation of these algorithms leveraging the simulator was performed, comparing the performance of both approaches under several network topologies.

The obtained results suggest that the configuration parameters introduced by our extensions to the algorithms have a significant impact on their performance. As such, a sensitivity analysis and evaluation with the objective of optimizing these configurations would be a relevant next step. As further future work, we could also consider other algorithms or variants of the algorithms studied here, such as the extension of Flow Updating to dynamic networks described in [4]. More generally, adding the possibility of nodes joining and leaving, and of links being added and removed, would be an useful addition both to the simulator and to the evaluation conducted here. Similarly, evaluating the ability of the algorithms to adapt to evolving input values would be relevant.

REFERENCES

[1] Albert-László Barabási and Réka Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512. https://doi.org/10.1126/science.286.5439.509

[2] Paul Erdős and Alfréd Rényi. 1959. On random graphs I. *Publicationes Mathematicae* 6 (1959), 290–297.

[3] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2009. Fault-Tolerant Aggregation by Flow Updating. In *Proceedings of the 9th IFIP International Conference on Distributed Applications and Interoperable Systems*. 73–86. https://doi.org/10.1007/978-3-642-02164-0_6

[4] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2010. Fault-Tolerant Aggregation for Dynamic Networks. In *Proceedings of the 29th IEEE Symposium on Reliable Distributed Systems*. 37–43. https://doi.org/10.1109/SRDS.2010.13

[5] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. 2015. A Survey of Distributed Data Aggregation Algorithms. *IEEE Communications Surveys & Tutorials* 17, 1 (2015), 381–404. https://doi.org/10.1109/COMST.2014.2354398

[6] David Kempe, Alin Dobra, and Johannes Gehrke. 2003. Gossip-Based Computation of Aggregate Information. In *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science*. 482–491. https://doi.org/10.1109/SFCS.2003.1238221

[7] Duncan Watts and Steven Strogatz. 1998. Collective dynamics of ‘small-world’ networks. *Nature* 393, 6684 (1998), 440–442. https://doi.org/10.1038/30918