Near-Optimal Communication-Time Tradeoff in Fault-Tolerant Computation of Aggregate Functions

Yuda Zhao
National University of Singapore
Republic of Singapore
yuda@comp.nus.edu.sg

Haifeng Yu
National University of Singapore
Republic of Singapore
haifeng@comp.nus.edu.sg

Binbin Chen
Advanced Digital Sciences Center
Republic of Singapore
binbin.chen@adsc.com.sg

ABSTRACT

This paper considers the problem of computing general *commutative and associative aggregate functions* (such as SUM) over distributed inputs held by nodes in a distributed system, while tolerating failures. Specifically, there are N nodes in the system, and the topology among them is modeled as a general undirected graph. Whenever a node sends a message, the message is received by all of its neighbors in the graph. Each node has an input, and the goal is for a special *root* node (e.g., the base station in wireless sensor networks or the gateway node in wireless ad hoc networks) to learn a certain commutative and associate aggregate of all these inputs. All nodes in the system except the root node may experience crash failures, with the total number of edges incidental to failed nodes being upper bounded by f. The timing model is synchronous where protocols proceed in rounds. Within such a context, we focus on the following question:

Under any given constraint on time complexity, what is the lowest communication complexity, in terms of the number of bits sent (i.e., locally broadcast) by each node, needed for computing general commutative and associate aggregate functions?

This work, for the first time, reduces the gap between the upper bound and the lower bound for the above question from *polynomial* to *polylog*. To achieve this reduction, we present significant improvements over both the existing upper bounds and the existing lower bounds on the problem.

Categories and Subject Descriptors

F.1.3 [**Computation by Abstract Devices**]: Complexity Measures and Classes; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

Keywords

Communication complexity, time complexity, communication-time tradeoff, fault tolerance, aggregate functions

1. INTRODUCTION

The problem of fault-tolerant aggregation. In recent years, there has been a line of research (e.g., [1, 4–6, 8, 9, 13, 14, 16, 17]) on computing aggregates over distributed inputs held by nodes in a distributed system. This paper focuses on the following specific fault-tolerant version of the problem (formally defined in Section 2): There are N nodes in the system, and the topology among them is modeled as a general undirected graph. Whenever a node sends a message, the message is received by all of its neighbors in the graph. (In other words, each "send" is a local broadcast.) Each node has a non-negative integer input that is no larger than some polynomial of N. The goal is for a special *root* node to compute a certain aggregate function over all these inputs. For example, the root can be the base station in wireless sensor networks or the gateway node in wireless ad hoc networks. We will focus on the SUM function first, and then trivially generalize to arbitrary *commutative and associative aggregate functions* (or CAAFs in short — see definition in Section 2). All nodes in the system except the root may experience crash failures. For convenience, we say that an edge *fails*, iff at least one of its end points experiences a crash failure. We use f to denote an upper bound on the total number of edge failures. We consider a synchronous timing model where protocols proceed in *rounds*.

We consider randomized protocols for computing SUM (or general CAAFs) that always generate a *correct* result. A sum result is *correct* [1] iff it falls between the sum of the inputs of all nodes and the sum of the inputs of all nodes that are still alive and are not partitioned from the root at the end of the protocol's execution.[1] We similarly define result *correctness* for general CAAFs. The *time complexity* (TC) of a protocol is defined to be the number (denoted as b) of *flooding rounds* needed for the protocol to terminate. Here each flooding round consists of d rounds where d is the diameter of the network. The *communication complexity* (CC) of a protocol is the maximum number of bits that a node needs to send (i.e., locally broadcast) in the entire execution of the protocol. Here the maximum is taken across all nodes in the system. Given such a context, this paper focuses on the following question:

Under any given constraint on TC, what is the lowest CC needed for computing SUM *(or general CAAFs)?*

Existing lower/upper bounds. The only known non-trivial lower bound so far on the CC of SUM protocols, in the fault-tolerant setting, was from our own previous work [4]. Specifically, there we

[1]For example, if a node fails or gets partitioned from the root (due to the failure of other nodes) right before the SUM protocol starts, incorporating the node's input into the final sum would not be possible.

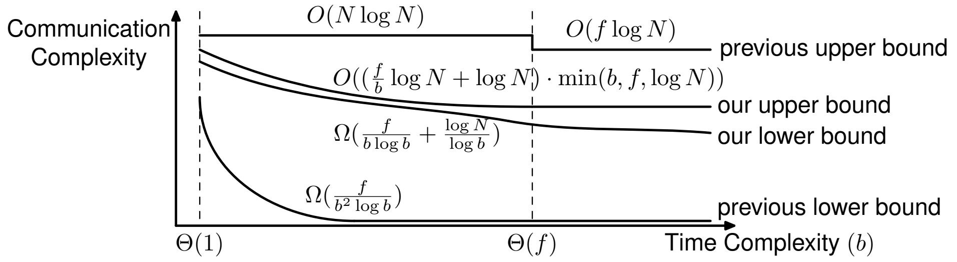

Figure 1: Summary of results on the SUM problem. Here b is the time complexity, and f is an upper bound on the total number of edges incident to failed nodes. Since the communication complexity depends on three parameters b, f, and N, the two-dimensional curves here are for illustration purposes only. Note that $O((\frac{f}{b} \log N + \log N) \cdot \min(b, f, \log N)) = O(\frac{f}{b} \log^2 N + \log^2 N)$.

proved that the CC is lower bounded by $\Omega(\frac{f}{b^2 \log b})$.[2] This lower bound indicates that the CC might be able to decrease polynomially with the TC (i.e., with b). There have been only a few upper bounds on the problem. Well-known tree-aggregation protocols [12] for computing SUM cannot tolerate failures. A brute-force SUM protocol, which has every node flood its id together with its value to the whole network, can tolerate arbitrary number of failures, while incurring $O(1)$ TC and $O(N \log N)$ CC. In comparison, under such TC, the current lower bound of $\Omega(\frac{f}{b^2 \log b})$ on CC is only $\Omega(f)$. There is also a folklore SUM protocol that tolerates failures by repeatedly invoking the naive tree-aggregation protocol until it experiences a failure-free run. This incurs $O(f)$ TC and $O(f \log N)$ CC. Under such TC, the current lower bound of $\Omega(\frac{f}{b^2 \log b})$ on CC is $\Omega(\frac{1}{f \log f})$. To summarize, the two fault-tolerant SUM protocols here both have fixed TC, and preclude the possibility of trading off TC with CC. Furthermore, even under that fixed TC, their CC is still a *polynomial* factor away from the lower bound (Figure 1).

Researchers have also studied the SUM problem when bounded errors are allowed in the final answer (e.g., [1, 4, 5, 8, 13, 14]), in either fault-tolerant or failure-free setting. Those results and their approaches are less related to this work. Nevertheless even there, in the fault-tolerant setting, a *polynomial* gap exists between the upper bound and lower bound as long as $b \geq 2$ [4].

For protocols that can compute general CAAFs, obviously existing lower bounds on SUM protocols directly carry over, since those protocols need to at least be able to compute SUM. It happens that the two existing (zero-error) SUM upper bound protocols work, without modification, for general CAAFs as well. We are not aware of any better lower/upper bounds on zero-error protocols for general CAAFs.

There have also been research efforts (e.g., [6, 9, 16, 17]) on computing aggregate functions (such as SELECTION) that are not CAAFs. But none of these efforts consider failures. Finally, some researchers (e.g., [10]) have studied the token dissemination problem in *dynamic networks*. In comparison, this paper considers i) the aggregation problem where the "tokens" can be aggregated, and ii) node failures in static networks. With these differences, their techniques/results have limited relevance to our setting.

Our results. This work reduces the gap between the upper and lower bound for the above SUM problem from *polynomial* to *polylog*, over the entire time complexity spectrum. Improvements over both the existing upper bound and lower bound turn out to be necessary to achieve this significant reduction. Specifically, we present a novel upper bound of $O(\frac{f}{b} \log^2 N + \log^2 N)$ as well as a novel lower bound of $\Omega(\frac{f}{b \log b} + \frac{\log N}{\log b})$, for the CC of SUM protocols whose TC is within b flooding rounds (Figure 1)[3]. Note that our upper bound is at most $\log^2 N \log b$ factor away from our improved lower bound.

Our upper bound protocol also, for the first time, allows a tunable tradeoff between CC and TC where the CC can decrease polynomially with the TC. Using the standard doubling trick, the full version of this paper [18] further shows that the protocol can be easily extended to settings with unknown f, while only increasing the CC by a $\log N$ factor. Doing so will achieve a property similar to *early termination* — namely, the overhead of the protocol will automatically vary depending on the actual number of failures occurred during its execution.

Finally, same as some existing SUM protocols, our SUM protocol and its guarantees trivially generalizes to arbitrary CAAFs as well. This gives an $O(\frac{f}{b} \log^2 N + \log^2 N)$ upper bound on general CAAFs. By our lower bound, within polylog factors, this upper bound is the best that one can hope for.

Our main techniques. Our upper bound is non-trivial and involves the synthesis of two novel building blocks:

- We first propose a novel deterministic aggregation protocol AGG, parameterized by $t \geq 0$, with TC of $O(1)$ flooding rounds and CC of $O((t+1) \log N)$ bits. If the actual number of edge failures is no more than t, AGG always generates a correct result. Note that setting $t = f$ directly gives us $O(1)$ TC and $O(f \log N)$ CC, which is already much better than the two existing SUM protocols mentioned earlier. The key technique in AGG is to take *speculative* actions to save time, instead of waiting for failures to be detected and then falling back to a second plan. We further carefully design a distributed mechanism to determine which speculative actions' effects should be retained or discarded, while using only local information.

- If the number of edge failures exceeds t, AGG may *unknowingly* generates a wrong result. We hence design a novel deterministic distributed verification protocol VERI, which aims to tell whether AGG's result is correct. VERI is also parameterized by t and incurs $O(1)$ TC and $O((t+1) \log N)$ CC. The key technique in VERI is that we allow it to have one-sided error. Specifically, we allow VERI to sometimes

[2]The model in [4] slightly differs from the model in this paper. But the results there can still be trivially adapted to this paper. Such trivial adaptation will be rigorously described in the full version of this paper [18].

[3]We have actually proved an upper bound of $O((\frac{f}{b} \log N + \log N) \cdot \min(b, f, \log N))$. But for clarity, this paper uses the simpler form of $O(\frac{f}{b} \log^2 N + \log^2 N)$ in most places. The main novelty in our lower bound is the $\frac{f}{b \log b}$ term. The $\frac{\log N}{\log b}$ term comes, in a relatively straightforward way, from applying the results in [7] to the output domain size of $\Omega(N)$.

N	number of nodes in the system		b	SUM protocol's TC, in terms of flooding rounds
n	size of two-party problems		d	diameter of the topology $\mathcal{G}$
f	upper bound on the number of edge failures		c	diameter of the topology never exceeds cd due to failures
t	parameter in AGG and VERI		l	a node's level in the aggregation tree

Table 1: Key notations.

err when AGG does not err. VERI also employs a similar distributed mechanism to the one in AGG to avoid the need for global information in its execution.

Our upper bound protocol is eventually obtained by executing multiple pairs of AGG and VERI in a proper way.

Our new lower bound builds upon our previous lower bound [4], which was obtained via information cost arguments. To obtain this new result, we first introduce a new two-party problem EQUALITYCP, and then leverage a strong result on the Sperner capacity of general directed graphs [3] (instead of relying on information cost arguments). The possibility of leveraging the Sperner capacity of the cyclic q-gon [2] was hinted in a single footnote, but without any further details or any final result, in our previous paper [4]. This paper not only presents the specific lower bound obtained via that approach, but also slightly strengthens that approach — The approach in this paper yields a slightly better constant (specifically in Lemma 11) than the originally hinted approach.

2. MODEL AND DEFINITIONS

Commutative and associative aggregate functions. A binary operator $\diamond$ is *commutative and associative* if for all operands o_1, o_2, and o_3, we have $o_1 \diamond o_2 = o_2 \diamond o_1$ and $(o_1 \diamond o_2) \diamond o_3 = o_1 \diamond (o_2 \diamond o_3)$. A function $\mathcal{F}$ is called a *commutative and associative aggregate function*, or *CAAF* in short, if i) there exists a commutative and associative binary operator $\diamond$ such that $\mathcal{F}(o_1, o_2, ..., o_N) = o_1 \diamond o_2 \diamond ... \diamond o_N$, and ii) the domain size of $o_{i_1} \diamond o_{i_2} \diamond ... \diamond o_{i_k}$ is at most polynomial with respect to N, for all $1 \leq k \leq N$ where i_1 through i_k are arbitrary distinct indices. The second requirement stems from the "aggregate" nature of the function — "aggregating" o_{i_1} through o_{i_k} should generate a concise output. CAAF covers a wide range of common aggregate functions such as SUM and COUNT. Many other aggregate functions such as AVERAGE, MEDIAN, and SELECTION are related to CAAFs. For example, it is known [16] that MEDIAN and SELECTION can be solved using COUNT by doing a binary search over the output domain.

Our novel upper bound applies to all CAAFs. But for the sake of clarity, the rest of the paper will prove the upper bound only for SUM. This allows us to conveniently use natural phrases such as "the sum of these 4 inputs". None of our arguments there rely on the specifics of the addition operator. Hence generalizing to other CAAFs is entirely trivial – one only needs to replace the addition operator with $\diamond$.

Network model. There are N nodes in the system, where N is known by the protocol. (See Table 1 for notation summary.) Each node has a unique id of $\log N$ bits (log in this paper is always base 2). Node i has an integer *input* o_i, whose domain size is polynomial of N. The goal is for a special *root* node, whose id is known by all nodes, to learn the sum of all these inputs. The topology among the N nodes is modeled as an undirected graph $\mathcal{G}$. A node knows neither $\mathcal{G}$ nor its neighbors in $\mathcal{G}$. We impose no restriction on $\mathcal{G}$ except that it needs to be connected. We consider a synchronous timing model where protocols proceed in *rounds*. In each round, each node first receives all the messages sent by its neighbors in $\mathcal{G}$ in the previous round. Next it does some local computation and then may choose to send (i.e., locally broadcast) a single message, which will be received by all its neighbors in $\mathcal{G}$ in the next round. To make our results as strong as possible, we assume that all nodes start execution at round 1 for our lower bound. For our upper bound, we assume that the root initiates the protocol at round 1. Upon receiving the first message, a non-root node gets "activated" and joins the execution.

Failure model. All nodes in the system, except the root, may experience crash failures. A node that is disconnected from the root (i.e., has no path to the root) due to the failures of other nodes is also considered as failed. We consider only oblivious failure adversaries that adversarially decide beforehand (i.e., before the protocol flips any coins) which nodes fail at what time. For convenience, we say that an edge *fails*, iff at least one of its end points experiences a crash failure. We use f to denote an upper bound on the total number of edge failures, ranging from 1 to $\Theta(N)$.[4] We assume that f is known to the protocol.[5]

Let s_2 be the set of the inputs of all nodes, and s_1 be the set of the inputs of all nodes that have not failed by the end of the protocol's execution. Following [1, 4], we say that a sum result is *correct* if it is in the interval of $[\sum_{o \in s_1} o, \sum_{o \in s_2} o]$. We naturally generalize such result *correctness* definition to any CAAF: Here the result is *correct* if it is between $\min_{s_1 \subseteq s \subseteq s_2}(\diamond_{o \in s} o)$ and $\max_{s_1 \subseteq s \subseteq s_2}(\diamond_{o \in s} o)$.[6] We only consider randomized protocols for computing SUM (or general CAAFs) that always generate a correct result.

Time complexity and communication complexity. Most of the definitions here directly follow from [4]. To make our results as strong as possible, our upper bound only uses private coins, while our lower bound allows public coins.

With respect to a topology $\mathcal{G}$, the *time complexity* (TC) of a SUM protocol describes the number of rounds needed for it to terminate, under the worst-case inputs of nodes in $\mathcal{G}$, the worst-case failure adversary (parameterized by f), and the worst-case coin flips. The shape of $\mathcal{G}$ has a large impact on TC. Hence similar to [4], we will always describe TC in terms of *flooding rounds*. Here each flooding round consists of d rounds, where d is $\mathcal{G}$'s diameter and is assumed to be known to the protocol. We use b to denote the TC in terms of flooding rounds (i.e., the total number of rounds would be bd).

At any given point of time between round 1 and round bd, let $\mathcal{H}$ be the same as $\mathcal{G}$ except that all the failed nodes and their incidental edges have been deleted. $\mathcal{H}$'s diameter may be larger or smaller than $\mathcal{G}$. For a flooding round to remain meaningful in such a context, we assume that the failures do not substantially increase the network's diameter. Specifically, we assume that the diameter of $\mathcal{H}$ is no larger than $c \cdot d$, where c is some constant known to the protocol. As part of our future work, we are currently working on

[4]Certain graphs may have more than $\Theta(N)$ edges. But we focus on f between 1 and $\Theta(N)$ which applies to all graphs.

[5]The full version of this paper [18] explains how to remove this assumption using a simple doubling trick.

[6]Alternatively, one could define a result to be *correct* iff the result equals $\diamond_{o \in s} o$ for some s where $s_1 \subseteq s \subseteq s_2$. All our theorems and proofs hold, without any modification, under such an alternative definition.

Algorithm 1 Our upper bound protocol. Here b, c, and f are input parameters with $b \geq 21c$.

```
1: $x = \lfloor \frac{b-2c}{19c} \rfloor$; the root uses private coins to select $\log N$ integers, with replacement, from the range of $[1, x]$;
   let the selected integers be $y_1, y_2, ..., y_{\log N}$, in non-decreasing order;
2: for all integer $i \in [1, \log N]$ where ($i = 1$ or $y_i \neq y_{i-1}$) do
3:     at the beginning of flooding round $((y_i - 1) \times 19c + 1)$, root initiates a pair of AGG and VERI executions, both with $t = \lfloor \frac{2f}{x} \rfloor$;
       // this pair of executions will end by flooding round $(y_i \times 19c)$;
4:     if (AGG does not abort and VERI outputs true) then output AGG's result and terminate;
5: end for
6: at the beginning of the last $2c$ flooding rounds, root initiates the brute-force SUM protocol, outputs its result, and terminates;
```

a new lower bound proof that aims to show the necessity of this requirement, which is however beyond the scope of this paper.

With respect to $\mathcal{G}$, we define a node i's *communication complexity* (denoted as a_i) when running a SUM protocol to be the total number of bits it sends (i.e., locally broadcasts) when running the protocol, under the worst-case inputs of nodes in $\mathcal{G}$, the worst-case failure adversary (parameterized by f), and the average-case coin flips. A SUM protocol's *communication complexity* (CC) is the maximum a_i across all i's. Note that here we define CC over the bottleneck node instead of over the average node, which is appropriate in our distributed setting with a general topology and consistent with prior work [16].

Let $a_{\mathcal{G}}$ be the smallest CC under the topology $\mathcal{G}$ with at most f edge failures, across all SUM protocols whose TC is at most b flooding rounds. We define $\text{FT}_0(\text{SUM}_N, f, b)$ to be the maximum $a_{\mathcal{G}}$ across all $\mathcal{G}$ where $\mathcal{G}$ is connected and has exactly N nodes. Here "FT" stands for "fault-tolerant" and the subscript "0" stands for "zero-error".

Communication complexity of two-party problems. In Section 7, we will need to reason about the communication complexity of certain two-party problems. In those problems, Alice and Bob have inputs X and Y respectively, and they aim to compute a certain function $\Pi(X, Y)$. We only require Alice to learn the final result. We use n to denote the size of these two-party problems (with N being reserved to denote the number of nodes in $\mathcal{G}$). Unless otherwise noted, by a protocol for solving Π, we mean a public coin Las Vegas protocol. We define the *communication complexity* (CC) of Π (denoted as $\mathcal{R}_0(\Pi)$) to be smallest expected (with expectation taken over the coin flips) number of bits sent by Alice and Bob combined, across all protocols for solving Π.

3. SUMMARY OF RESULTS

The following two theorems summarize our main results:

THEOREM 1. *For any $b \geq 21c$ and $1 \leq f \leq N$, we have:*

$$\begin{aligned}\text{FT}_0(\text{SUM}_N, f, b) &= O((\frac{f}{b}\log N + \log N) \cdot \min(b, f, \log N)) \\ &= O(\frac{f}{b}\log^2 N + \log^2 N).\end{aligned}$$

THEOREM 2. *For any $b \geq 1$ and $1 \leq f \leq N$, we have:*

$$\text{FT}_0(\text{SUM}_N, f, b) = \Omega(\frac{f}{b \log b} + \frac{\log N}{\log b}).$$

The rest of the paper proves the two theorems above. Here we give an overview of the structure of our upper bound protocol (Algorithm 1) that is used to prove Theorem 1. Given total b flooding rounds as a constraint on TC, we divide the first $b - 2c$ flooding rounds into $x = \Theta(b)$ *intervals*, with each interval having $19c = \Theta(1)$ flooding rounds. Thanks to the small TC of AGG and VERI, running AGG followed by VERI will take at most one interval. If the edge failures were evenly distributed across all the intervals, then each interval would have at most $\frac{f}{x}$ edge failures. In such a case, running AGG parameterized with $t = \frac{f}{x}$ in any single interval would already produce a correct result, while incurring a desirable CC of $O((\frac{f}{b} + 1) \log N)$. Here recall that t is the number of edge failures that AGG intends to tolerate, and the CC of AGG is $O((t + 1) \log N)$.

Since the edge failures are not always evenly distributed, we need a more complex design. Specifically, the root uses private coins to select $\log N$ intervals uniformly randomly. In each selected interval, the root initiates a pair of AGG and VERI executions, both with $t = \lfloor \frac{2f}{x} \rfloor$. One can easily see that with probability at least $\frac{1}{2}$, a random interval has no more than t edge failures. Hence with probability at least $1 - \frac{1}{N}$, the number of edge failures in *some* selected interval is small enough for AGG to tolerate. But if there have been more than t edge failures in an interval, then AGG may *unknowingly* produce a wrong result. A difficulty here is that we cannot easily determine the number of edge failures that have occurred in a given interval, since it involves counting while tolerating potential additional failures during counting. Hence instead of checking the number of edge failures in a given interval, our protocol invokes VERI after AGG, and then checks the condition at Line 4 of Algorithm 1. If the condition is met, the protocol outputs AGG's result and terminates. By Theorem 5 and 7 later, such a result must be correct. Furthermore by Theorem 4 and 7 later, if the number of edge failures in an interval is no more than t, then the condition at Line 4 is guaranteed to be met.

Having given an intuitive overview on the protocol's correctness, we move on to look at its CC. Since there can be at most x intervals in total and f intervals with failures, AGG and VERI will be executed at most $\min(x, f + 1, \log N)$ times. The CC incurred by each AGG and VERI invocation is $O((t + 1) \log N)$ bits, resulting in total $O((\frac{f}{b} \log N + \log N) \cdot \min(b, f, \log N))$ bits in all the intervals. Next, the probability of reaching Line 6 is at most $\frac{1}{N}$. As explained in Section 1, the CC of the brute-force SUM protocol is $O(N \log N)$. Hence the CC incurred at Line 6, over average coin-flips, is $O(\log N)$.

Next in Section 4 and 5, we focus on AGG and VERI, and prove their properties. Section 6 then provides the full proof for Theorem 1. Theorem 2 will be discussed in Section 7.

4. THE AGG PROTOCOL

Overview. Algorithm 2 at the end of this paper provides the pseudocode for AGG. AGG has an input parameter t ($t \geq 0$), which is the number of edge failures that it intends to tolerate. When running AGG, a node will flood[7] a special symbol to abort AGG once it has sent $(11t + 14)(\log N + 5)$ bits. Such a mechanism will never be

[7]Throughout this paper, a node *floods* a certain message by first sending the message to its neighbors, and then the other nodes simply forward that message upon first receiving it.

triggered (as we prove later) if the actual number of edge failures is no larger than t. If the actual number of edge failures exceeds t, aborting before the CC gets too large enables AGG to properly bound its CC.

AGG first constructs a spanning tree and does a standard tree-based aggregation, where each non-root node sends its *partial sum* upstream (i.e., towards the root) along the tree. The *partial sum* of a node (either non-root or root) is the sum of the node's own input and all the partial sums received from its children. A key impact of failures is that they may block and prevent certain partial sums from propagating upstream. If a partial sum from a node B is blocked, a natural solution is to have B flood its partial sum, since flooding has the maximum resilience against failures. If the flooding does reach the root, the root can then incorporate B's partial sum to the final result. A second thought, however, shows that even with flooding, B's partial sum may still fail to reach the root if B's entire neighborhood fails immediately after B initiates the flooding. When this happens, the system needs to fall back and flood the partial sums of B's children, or B's descendants if B's children have also failed.

The key challenge here is that we need to do this within $O(1)$ flooding rounds. We cannot afford to wait to see whether B's partial sums get successfully flooded, and then fall back to flooding some other partial sums if things did not go well. To save time, we will have to do floodings *speculatively*, before knowing which floodings will be needed. This in turn leads to a second challenge: There will be overlap (or duplicates) in the partial sums received by the root (e.g., partial sums from both B and some of B's descendants). We need a careful mechanism to avoid double counting, which is non-trivial, especially without global knowledge about the tree topology.

The following sections present the details of AGG. At a high-level, AGG has 3 sequential phases: i) spanning tree construction and tree-aggregation (Section 4.1), ii) identifying potentially blocked partial sums and (speculatively) flooding them (Section 4.2), and iii) using a distributed mechanism based on *witnesses* to avoid double counting (Section 4.3). To facilitate understanding, the discussion here will be intuitive — we leave the formal proofs to the full version of this paper [18].

4.1 Tree Construction/Aggregation and Some Key Concepts

This section first describes the tree construction/aggregation phase in AGG, which is largely standard. Next we formalize a number of new concepts that are key for our later design.

Tree construction and aggregation. To construct the tree, the root first sends a tree_construct message, together with a hop count. A node B waits for the first tree_construct message it receives. Note that this message easily enables B to figure out the current round, and synchronize its round counter with the root. Let A denote the sender of that message. B sends an ack message indicating to A that B is A's child, and then sends a tree_construct message itself to continue constructing the tree. B's failing before sending ack will be equivalent to B not being present in the network. The failure of B after sending ack will be dealt with later in AGG. From now on in this paper, the notions of "parent", "child", "ancestor", and "descendant" will always be with respect to this tree.

Next AGG does standard tree-aggregation. Consider a given node B, and let l be its *level* (i.e., its distance from the root). Node B acts in the $(cd-l+1)$th round during tree-aggregation, by summing up its own input with all the partial sums received from its children so far, and then sending the new partial sum to B's parent. Note that B does not necessarily wait for a message from each of its children, since some may have failed. Each partial sum thus is the sum of inputs from a subset of the nodes, and we also say that the partial sum *includes* those inputs.

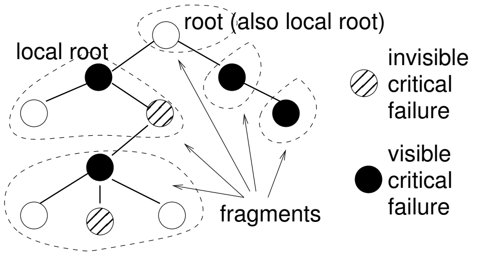

Figure 2: Example aggregation tree and fragments.

Some key concepts. We say that a node B at level l experiences a *critical failure* if it fails after sending ack during tree construction and before taking its action in the $(cd - l + 1)$th round during tree-aggregation. Such a critical failure can be easily detected by B's parent A (if A is alive) during that round, when A does not receive the scheduled message from B. We want critical failure to become global knowledge when possible. To do so, A will flood a message claiming that B experiences a critical failure. We say that a flooding is *successful* if the flooded message eventually reaches the root. One can easily see that a successful flooding must reach all live nodes within cd rounds. We say that a critical failure is *visible* if it is eventually seen by the root. Otherwise it is *invisible*. To help understanding, the next will first assume that all critical failures are visible, and then remove that assumption in Section 4.4.

Imagine that we remove all those edges connecting visible critical failures with their corresponding parents. Doing so partitions the aggregation tree into many smaller trees which we call *fragments* (Figure 2). A node's *local ancestors (descendants)* are all its ancestors (descendants) within the node's fragment. Each fragment also has its own *local root*. A fragment has a clean property: The partial sum of a node never includes inputs from nodes outside of its fragment, since those inputs have been blocked by the visible critical failures. Hence we can restrict most of our discussions to within a fragment.

A node A's partial sum is a *representative* of a node B iff i) A is either B itself or A is B's local ancestor, and ii) the tree path from A to B (excluding A and B) contains no invisible critical failures. Intuitively, B's representative must include B's input. A *representative set* is a set of partial sums with the following property: For any node B, if B is alive at (has failed by) the end of the VERI execution that immediately follows AGG, then a representative set contains exactly one (at most one) representative of B. Intuitively, if we obtain a representative set and sum up all the partial sums there, we get a correct sum result.

4.2 Identify and Flood Potentially Blocked Partial Sums

With the above notion of representative set, our goal in the remainder of AGG is for the root to obtain a representative set. If there were no critical failures at all, then the root's partial sum by itself is already a representative set. With critical failures, a representative set will contain not only the root's partial sum but also those blocked partial sums. Consider the example in Figure 3. Here, the root's partial sum, A's partial sum, and F's partial sum form a representative set. Imagine that we have A and F flood their partial sums, so that the root can get those and add those to the final result. However, A, B, and C all fail right before A intends to flood. Hence A's partial sum is lost and we now need D and E to flood their partial sums, which will form a second representative

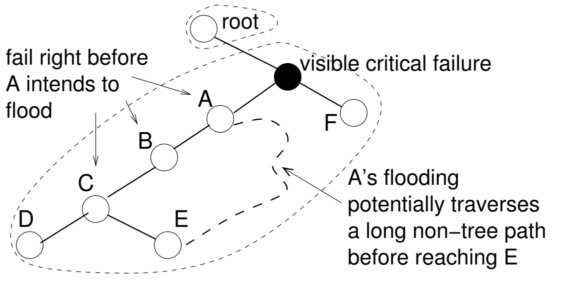
root
fail right before A intends to flood
visible critical failure
A
B
C
D
E
F
A's flooding potentially traverses a long non−tree path before reaching E

Figure 3: Why speculative flooding is needed.

set together with the root's and F's partial sum. Ideally, D and E should do so after they know that A's flooding has failed. Unfortunately, it can take one flooding round before such determination can be made, since A's flooding could traverse a long non-tree path before reaching E (Figure 3). Similarly, if E's flooding also fails, then E's local descendants (if any) may need to wait one more flooding round before taking action.

This example shows that to have small TC, nodes need to flood speculatively, before knowing that the flooding is needed. AGG uses the following elegant design to decide which node initiates flooding at what time: The root always floods its partial sum in the first round of the partial sum flooding phase. A non-root node B at level l floods its partial sum in the $(l + 1)$th round of the phase, iff in that round it does not receive any flooding message (containing any partial sums) from its parent A. Here A may or may not be the *initiator* of the corresponding flooding. This design has two important features. First, the design never does excessive floodings: If A has not failed by the $(l+1)$th round, B must receive some message from A and will not initiate its own flooding. This implies that the total number of floodings is linear with the number of edge failures. Second, the full version of this paper [18] proves that the design always floods a superset of those partial sums that need to be flooded. Namely, if B did not flood its own partial sum, then B must have forwarded a "better" partial sum that includes all those inputs included by B's partial sum.

4.3 Avoid Double Counting While Using Only Limited Information

The root potentially receives many flooded partial sums, and it needs to pick a representative set to avoid double counting. The partial sums seen by the root can be classified into three mutually-exclusive categories: The partial sum of a node B is *dominated* if the root also sees A's partial sum where A is B's local ancestor. A non-dominated partial sum of a node B is *compulsory* if either B or at least one of B's local descendants is still alive at the end of the VERI execution that immediately follows AGG, otherwise the non-dominated partial sum is called *optional*. The full version of this paper [18] proves that the union of all compulsory partial sums and any subset of optional partial sums forms a representative set.

Without knowledge of the global tree topology for labeling each partial sum, AGG maintains distributed topology information to do so. Specifically, when initially constructing the tree, AGG lets each node learn the ids of its nearest $2t$ ancestors. Interestingly, such limited information is already sufficient for AGG to select a representative set, in the following way via *witnesses*.

Having witnesses label partial sums. A node B's *witness* is either B itself or some local descendant of B whose distance to B is at most t. Let the local root of B's fragment be X and consider B's witness C. First, if C sees X among its $2t$ ancestors, then its $2t$ ancestors must contain all of B's local ancestors. Note that partial sums seen by the root must be seen by all live nodes as well. Hence if C sees a partial sum from some local ancestor of B's, C knows that B's partial sum is dominated and will thus flood its determination $\langle\text{dominated}, B\rangle$ to inform the root. Otherwise C floods its determination $\langle\text{compulsory}||\text{optional}, B\rangle$. When B has multiple witnesses, such determination may be flooded multiple times. This does not increase CC since all the determinations are identical, and a node only needs to participate in one such flooding.

Second, if C does not see X among its $2t$ ancestors, then there must be at least $2t - t = t$ nodes on the tree path from B to X (excluding B and X). If the number of edge failures is no more than t, then there must be at least one live node on that tree path between B and X. That live node must have successfully flooded a partial sum of either itself or one of its local ancestors. This implies that B's partial sum must be dominated. C will thus flood the determination $\langle\text{dominated}, B\rangle$. If the number of edge failures exceeds t, such determination might be wrong, which will be dealt with later by VERI.

Finally, it is possible for all of B's witnesses to fail (or for their floodings to fail to reach the root). In such a case, B's farthest local descendant must be no more than t hops away from B, since otherwise the number of edge failures will be more than t. (Again, the case where the number of edge failures exceeds t will be dealt with by VERI.) This implies that B and its local descendants must have all failed, since they are all B's witnesses. Hence if the root does not receive any determination on B's partial sum, the root knows that the partial sum cannot be compulsory, and must be either dominated or optional.

Take all three cases into account, to form a representative set, the root simply includes in the set a partial sum from a node B iff $\langle\text{compulsory}||\text{optional}, B\rangle$ has been received.

4.4 Complexity and Correctness of AGG

We have been assuming no invisible critical failures. The full version of this paper [18] proves that all the local ancestors of an invisible critical failure must have failed by the end of the tree-aggregation phase. Leveraging such observation, there we show that invisible critical failures does not affect any of our arguments earlier.

The following two theorems prove the complexity and correctness guarantees of AGG:

THEOREM 3. *The time complexity and communication complexity of* AGG *are no more than* $11c$ *flooding rounds and* $O((t + 1)\log N)$ *bits, respectively.*

Proof: The pseudo-code in Algorithm 2 obviously shows that AGG terminates within $7cd + 4$ rounds, which are at most $11c$ flooding rounds. For communication complexity, recall that in AGG, a node will flood a special symbol to abort AGG once it has sent $(11t + 14)(\log N + 5)$ bits. □

THEOREM 4. *If there are at most* t *edge failures during the execution of* AGG, *then* AGG *never aborts and always outputs a correct result.*

Proof: See the full version of this paper [18]. □

5. THE VERI PROTOCOL

We again focus on intuitions here, and leave the formal proofs to the full version of this paper [18].

Overview. Algorithm 3 at the end of this paper provides the pseudo-code for VERI. VERI aims to determine whether AGG's output is correct. The natural approach is for VERI to determine whether

scenario	AGG	VERI
1. no more than t edge failures (implying no LFC)	output correct result	output `true`
2. more than t edge failures and no LFC	output correct result or abort	no guarantee
3. more than t edge failures and exists LFC	no guarantee	output `false`

Table 2: Guarantees of AGG and VERI under different scenarios.

there have been more than t edge failures. This turns out to be difficult since it involves counting while tolerating potential additional failures during counting. Instead, our approach is to i) identify a weaker requirement that is nevertheless sufficient for AGG not to err, and ii) allow VERI to sometimes err when AGG does not err. Such a weaker requirement on VERI eventually makes an efficient design possible.

Specifically, with respect to a pair of AGG and VERI execution (both with parameter t), a *long failure chain* (LFC) is a chain of t nodes $A_1, A_2, ..., A_t$ within the same fragment such that i) A_i is the parent of A_{i+1} ($1 \le i \le t-1$), ii) all of them have failed by the end of the AGG execution, and iii) A_t has at least one local descendant that is alive at the end of the VERI execution. Here the notions of fragment, parent, and etc are all defined based on the AGG execution. A_1 and A_t are called the *head* and *tail* of the LFC, respectively. Note that having no more than t edge failures implies no LFC, while the reverse is not true. The following theorem claims that regardless of the number of edge failures, AGG will not err as long as there is no LFC.

THEOREM 5. *If there is no LFC, then* AGG *either outputs a correct result or aborts.*

Proof: See the full version of this paper [18]. □

The theorem implies that VERI may safely err in the 2nd scenario in Table 2, where there are more than t edge failures but no LFC. Table 2 also summarizes the guarantees of AGG and VERI in all other possible scenarios.

5.1 Design of The VERI Protocol

By the above discussion, we design VERI by focusing on detecting LFCs. Similar to AGG, in VERI once a node has sent $(5t+7)(3\log N+10)$ bits, it will flood a special symbol to cause VERI to output `false`.

Strawman design assuming no additional failures. To help understanding, we first describe a strawman design while assuming that there are no additional failures occurring during VERI's execution. A simple way to detect LFCs is for each node to ping its parent and children on the (aggregation) tree, and to flood the information about detected failures to all other nodes. Those *failed parents* and *failed children* are potentially tails and heads of LFCs. Without knowing the global tree topology, we will leverage the same witnesses as in Section 4.3 to determine whether they are indeed tails and heads of LFCs. Consider a failed parent B and a witness C of B's. Recall that B's witness is either B itself or some local descendant of B whose distance to B is at most t. C finds, among its $2t$ ancestors, B's nearest ancestor A such that A is either a failed child or a fragment boundary. One can easily see that B is the tail of an LFC iff A is at least $t-1$ hops away from B. Thus C can precisely determine whether B is the tail of an LFC, and can flood such determination to inform the root.

Failures of the witnesses. We now move on to the actual VERI design, by explaining how different kinds of failures during VERI's execution are addressed. We first consider the failures of the witnesses: In the earlier example, it is possible for all of B's witnesses to fail, so that no node can make a proper determination. We overcome this key challenge precisely by allowing VERI to err, as explained below.

First, we need AGG to maintain some additional information: During AGG's aggregation phase, we have each node learn the maximum level among its local descendants. This can be easily done by having nodes propagate upstream, along with the partial sum, the maximum level it has seen among its local descendants. Now in VERI, imagine that we can infer the distance x from B to B's farthest local descendants.[8] If the root does not receive any determination on whether B is the tail of some LFC (implying that all of B's witnesses have failed), the root applies the following rule: If $x \le t$, it claims that B is not the tail of an LFC. Otherwise it claims that B is the tail of an LFC, and outputs `false`.

To see when the above rule gives a correct/wrong determination, we separately consider two cases. First, $x \le t$ implies that all of B's local descendants are B's witnesses. They must have all failed since all witnesses have failed. In turn, by definition B must not be the tail of an LFC. Second, $x > t$ implies that B has at least t witnesses and all of them have failed. We still cannot determine whether there exists an LFC. But since VERI is allowed to make one-sided error when there are more than t edge failures (i.e., the 2nd and 3rd scenario in Table 2), VERI can simply output `false` in such a case.

When to detect failures. We move on to consider additional failures during the detection of failed parents/children. Those failures may prevent the floodings of information about failed parents/children from reaching the root. This is similar to flooded partial sums getting lost in Section 4.2 and Figure 3. To deal with this, VERI uses the following design similar to the one in AGG: The root floods a single bit. If a node at level l does not receive this bit or any message (claiming the detection of failed parents) from its own parent within $l+1$ rounds, it floods a message claiming that its own parent is a failed parent. If B is the tail of an LFC, such design guarantees (see the full version of this paper [18] for a proof) to inform the root that either B or some of B's local descendant is a failed parent.

Detection of failed children is similarly done by propagating a single bit upstream along all the tree edges. Finally, VERI always detects failed parents first and then detects failed children. This is necessary for correctness, if additional failures may occur during VERI. We leave the details on how this ordering is leveraged in our proofs to the full version of this paper [18].

5.2 Complexity and Correctness of VERI

THEOREM 6. *The time complexity and communication complexity of* VERI *are no more than $8c$ flooding rounds and $O((t+1)\log N)$ bits, respectively.*

Proof: The pseudo-code in Algorithm 3 clearly shows that VERI always terminates within $5cd+3$ rounds, which are at most $8c$

[8]Since B may have failed early on, we may not be able to actually get x. Nevertheless, one can achieve a similar functionality by using the maximum level information from B's descendants. See the full version of this paper [18] for details.

flooding rounds. For communication complexity, recall that in VERI, a node will flood a special symbol to terminate VERI once it has sent over $(5t+7)(10+3\log N)$ bits. □

THEOREM 7. *Consider a pair of* AGG *and* VERI *execution, both parameterized by t. If there exists an LFC, then* VERI *must output* `false`. *If there are at most t edge failures, then* VERI *must output* `true`.

Proof: See the full version of this paper [18].□

6. PROOF FOR THEOREM 1

THEOREM 1 (RESTATED). *For any $b \geq 21c$ and $1 \leq f \leq N$, we have:*

$$\begin{aligned}\mathrm{FT}_0(\mathrm{SUM}_N, f, b) &= O((\frac{f}{b}\log N + \log N)\cdot \min(b, f, \log N)) \\ &= O(\frac{f}{b}\log^2 N + \log^2 N).\end{aligned}$$

Proof: We prove the theorem by constructing an upper bound protocol as in Algorithm 1. The following proves the time complexity, communication complexity, and correctness of Algorithm 1.

For TC, Theorem 3 and 6 tell us that a pair of AGG and VERI executions take no more than $19c$ flooding rounds, and hence Line 3 of Algorithm 1 can complete in time. At Line 6, the root will flood a single bit to all nodes to initiate a brute-force protocol, taking c flooding rounds. Upon receiving this bit, a node floods its id and its input to all other nodes. Within c flooding rounds, the root is guaranteed to receive all flooded messages initiated by nodes that are still alive at the end of the protocol. The root then adds up the input for each id, and outputs the sum. Hence Line 6 takes at most $2c$ flooding rounds. Putting it all together, the time complexity of Algorithm 1 is no more than b flooding rounds.

For CC, by Theorem 4 and 7, if there are no more than $t = \lfloor \frac{2f}{x} \rfloor$ edge failures within an interval, then AGG will not abort and VERI will output `true`. This will then allow Algorithm 1 to terminate immediately after that interval at Line 4. Thus AGG and VERI will be executed at most $\min(x, f+1, \log N)$ times at Line 3 of Algorithm 1, since there are (i) total at most x intervals, (ii) at most f edge failures and hence at most $f+1$ intervals with failures, and (iii) at most $\log N$ intervals selected. By Theorem 3 and 6, the CC of AGG and VERI are both $O((t+1)\log N)$. Hence the total CC incurred at Line 3 is $O((t+1)\cdot \min(b, f, \log N)\cdot \log N)$.

Next consider the CC incurred at Line 6. Since there are at most f edge failures in all the x intervals, with probability at least $\frac{1}{2}$, a uniformly random interval contains no more than $\lfloor \frac{2f}{x} \rfloor$ edge failures. Hence with probability at least $\frac{1}{2}$, by Theorem 4 and 7, AGG will not abort and VERI will output `true`, causing Algorithm 1 to terminate in that interval. The probability of reaching Line 6 is thus at most $1/2^{\log N} = 1/N$. The brute-force protocol at Line 6 itself has a CC of $O(N\log N)$, implying that the CC (over average-case coin flips) incurred at Line 6 is at most $O(\frac{1}{N}\cdot N\log N) = O(\log N)$.

Putting everything together, the CC of Algorithm 1 is:

$$\begin{aligned} & O((t+1)\cdot \min(b, f, \log N)\cdot \log N) + O(\log N) \\ = \; & O((\frac{f}{b}\log N + \log N)\cdot \min(b, f, \log N)) \\ = \; & O(\frac{f}{b}\log^2 N + \log^2 N).\end{aligned}$$

Finally, we prove that Algorithm 1 always produces a correct sum result. If it outputs a sum at Line 6 from the brute-force protocol, the result is trivially correct. If it outputs the result generated by AGG, then we know that AGG did not abort and VERI outputted `true`. By Theorem 7, we know that there must have been no LFC. In turn by Theorem 5, we know that the result generated by AGG (if it did not abort) must be correct. □

7. A NEW $\Omega(\frac{f}{b\log b} + \frac{\log N}{\log b})$ LOWER BOUND ON THE CC OF SUM(THEOREM 2)

Review of previous lower bound. Our previous work [4] obtained a lower bound on SUM's CC by reducing a two-party communication complexity problem UNIONSIZECP to SUM. In UNIONSIZECP$_{n,q}$, Alice has a string X of length n as her input. Each character in the string is an integer in $[0, q-1]$ where $q \geq 2$. Bob similarly has a string Y as his input. X and Y satisfy the *cycle promise*,[9] in the sense that for all $1 \leq i \leq n$, either $Y_i = X_i$ or $Y_i = (X_i + 1) \bmod q$. Here X_i and Y_i are the ith character of X and Y respectively. Alice and Bob aim to determine the quantity $|\{i | X_i \neq 0 \text{ or } Y_i \neq 0\}|$. Our previous work [4] proved a lower bound of $\Omega(\frac{n}{q^2}) - O(\log n)$ on the CC of UNIONSIZECP$_{n,q}$, and then obtained a lower bound on SUM via a reduction from UNIONSIZECP. Trivially adapting that lower bound to the model in this paper gives us a lower bound of $\Omega(\frac{f}{b^2\log b})$ in this paper's setting.

Our new lower bound. This section presents a new lower bound of $\Omega(\frac{f}{b\log b} + \frac{\log N}{\log b})$ for SUM, and this factor-b improvement is necessary to bring down the gap between the upper and lower bound to polylog. The key to achieving this improvement is a stronger lower bound of $\Omega(\frac{n}{q}) - O(\log n)$ on UNIONSIZECP. This lower bound on UNIONSIZECP is almost tight, given the existing $O(\frac{n}{q}\log n + \log q)$ upper bound [4].

To obtain this lower bound on UNIONSIZECP, we introduce a new two-party problem called EQUALITYCP$_{n,q}$, which is the same as UNIONSIZECP$_{n,q}$ except that in EQUALITYCP$_{n,q}$, Alice and Bob aim to determine whether X equals Y. We are interested in EQUALITYCP$_{n,q}$ because its rectangular properties are easier to study. The following theorem establishes a reduction from EQUALITYCP to UNIONSIZECP, based on the following observation: Knowing the result of UNIONSIZECP, Alice and Bob can infer whether there exists j such that $X_j = q-1$ and $Y_j = 0$. If there exists such j, then $X \neq Y$ and we are done. Otherwise for $1 \leq i \leq n$, we must have $Y_i = X_i$ or $Y_i = X_i + 1$ (note that there is no longer "mod q"). This implies that $X = Y$ iff $\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} Y_i$.

THEOREM 8. $\mathcal{R}_0(\mathrm{EQUALITYCP}_{n,q}) \leq \mathcal{R}_0(\mathrm{UNIONSIZECP}_{n,q}) + O(\log q) + O(\log n)$.

Proof: To solve EQUALITYCP, Alice and Bob first invoke the oracle UNIONSIZECP protocol on their inputs X and Y. Bob next sends Alice $\sum_{i=1}^{n} Y_i$, using $\log n + \log q$ bits, and the occurrence count (denoted as z) of the character 0 in Y, using $\log n$ bits. Alice finally outputs that X equals Y iff $\sum_{i=1}^{n} X_i = \sum_{i=1}^{n} Y_i$ and UNIONSIZECP(X, Y) equals $n - z$.

To show the correctness of the above protocol, note that if $X = Y$, then the two conditions trivially hold. We next prove the reverse direction. Since UNIONSIZECP$(X, Y) = n - z$, for all i where $Y_i = 0$, we have $X_i = 0$. In turn, there does not exist i such that $X_i = q-1$ and $Y_i = 0$. With this additional property, together with the cycle promise, we know that for $1 \leq i \leq n$, either $Y_i = X_i$ or $Y_i = X_i + 1$ (note that there is no longer "mod q"). Hence X must equal to Y since otherwise $\sum_{i=1}^{n} Y_i$ would be larger than $\sum_{i=1}^{n} X_i$. □

[9]The cycle promise described here is called the "alternative form" of the cycle promise in [4].

Next we apply an existing strong result on the Sperner capacity of directed graphs [3] to obtain a lower bound on the CC of EQUALITYCP. That result was originally stated in the context of a directed coding graph, and the following instantiates it in our specific context:

THEOREM 9. (Adapted from Theorem 3.2 in [3].) *Let S be a subset of $\{0, 1, 2, ..., q-1\}^n$ with the following property: For all $V, W \in S$ where $V \neq W$, there i) exists i such that $V_i \neq W_i$ and $V_i \neq (W_i + 1)$ mod q, and ii) exists j such that $W_j \neq V_j$ and $W_j \neq (V_j + 1)$ mod q. Then $|S| \leq (rank(\mathbb{M}))^n$ for any $q \times q$ matrix $\mathbb{M}$, where $\mathbb{M}_{i,i} = 1$ for all i, $\mathbb{M}_{i,j} = 0$ for all $(j - i)$ mod $q \in \{2, 3, ..., q-1\}$, and all other entries in $\mathbb{M}$ (i.e., $\mathbb{M}_{1,2}$, $\mathbb{M}_{2,3}$, ..., $\mathbb{M}_{q-1,q}$, and $\mathbb{M}_{q,1}$) can be arbitrary real numbers.*

THEOREM 10.

$$\mathcal{R}_0(\text{EQUALITYCP}_{n,q}) \quad = \quad \Omega(\frac{n}{q} - \log n - \log\log q).$$

Proof: Our definition of $\mathcal{R}_0$ allows public coins and only requires Alice to know the result. We define $\mathcal{R}_0^{\text{pri}}$ to be the same as $\mathcal{R}_0$ except that only private coins are allowed and both Alice and Bob are required to know the result. Using arguments based on rectangles [11], Lemma 11 next proves that $\mathcal{R}_0^{\text{pri}}(\text{EQUALITYCP}_{n,q}) \geq \frac{n}{q-1}$. The theorem follows since i) only one bit is needed for Alice to inform Bob the result, and ii) a public coin protocol using k bits here can always be simulated via private coins while using $O(k + \log\log(q^n \cdot q^n)) = O(k + \log n + \log\log q)$ bits [15]. □

LEMMA 11. $\mathcal{R}_0^{\text{pri}}(\text{EQUALITYCP}_{n,q}) \geq \frac{n}{q-1}$.

Proof sketch: It is well known [11][10] that for any (partial) function $h : X \times Y \to \{0, 1\}$, $\mathcal{R}_0^{\text{pri}}(h) \geq N(h) \geq \log C^1(h)$. Here $N(h)$ is the non-deterministic communication complexity, and $C^1(h)$ is the smallest number of monochromatic rectangles needed to cover (possibly with intersections) all the 1-entries in the matrix corresponding to h. The matrix $\mathbb{Z}$ corresponding to $\text{EQUALITYCP}_{n,q}$ is a $q^n \times q^n$ matrix. All 1-entries in $\mathbb{Z}$ are on the main diagonal. The remainder of $\mathbb{Z}$ consists of 0-entries and undefined entries that correspond to input pairs not satisfying the cycle promise. In any given covering of all the 1-entries using monochromatic rectangles, consider any two 1-entries $\mathbb{Z}_{V,V}$ (i.e., the entry for $X = V$ and $Y = V$) and $\mathbb{Z}_{W,W}$ in any rectangle used in the covering. For the rectangle to be monochromatic, $\mathbb{Z}_{W,V}$ and $\mathbb{Z}_{V,W}$ must not be 0-entries and hence must be undefined entries. This means that there i) exists i such that $V_i \neq W_i$ and $V_i \neq (W_i + 1)$ mod q, and ii) exists j such that $W_j \neq V_j$ and $W_j \neq (V_j + 1)$ mod q.

Applying Theorem 9 tells us that the number of 1-entries in such a monochromatic rectangle is upper bounded by $(rank(\mathbb{M}))^n$ for any $q \times q$ matrix $\mathbb{M}$ satisfying the properties specified in Theorem 9. We want to find such an $\mathbb{M}$ with a small rank, by properly choosing the values of $\mathbb{M}_{1,2}$, $\mathbb{M}_{2,3}$, ..., $\mathbb{M}_{q-1,q}$, and $\mathbb{M}_{q,1}$. We set all of them to be -1. We claim that the rank of such an $\mathbb{M}$ is exactly $q - 1$. To see why, note that adding up all the q rows gives us an all-zero row, and hence $rank(\mathbb{M}) \leq q - 1$. It is also easy to verify that the first $q - 1$ rows are linearly independent. Hence $rank(\mathbb{M}) = q - 1$, implying that the number of 1-entries in a monochromatic rectangle of $\mathbb{Z}$ is upper bounded by $(q - 1)^n$. Finally, because the total number of 1-entries in $\mathbb{Z}$ is q^n, we have $\mathcal{R}_0^{\text{pri}}(\text{EQUALITYCP}_{n,q}) \geq \log(q^n/(q-1)^n) = n\log(1 + \frac{1}{q-1}) \geq \frac{n}{q-1}$. □

[10]The result was originally stated for functions, though it trivially applies to partial functions as well.

THEOREM 12. $\mathcal{R}_0(\text{UNIONSIZECP}_{n,q}) = \Omega(\frac{n}{q}) - O(\log n)$.

Proof: The equation trivially holds for $n \leq q$. For $n > q$, combining Theorem 8 and 10 directly yields the result. □

The $\Omega(\frac{f}{b \log b})$ term in Theorem 2 then follows naturally from Theorem 12 and the known reduction [4] from UNIONSIZECP to SUM. The extra $\Omega(\frac{\log N}{\log b})$ term in Theorem 2 comes from the $\Omega(N)$ domain size of the sum result. By results in [7], sending $\Omega(\log N)$ bits of information to the root within b flooding rounds (and hence within b rounds under the worst-case topology) requires sending $\Omega(\frac{\log N}{\log b})$ actual bits. We defer the full proof of Theorem 2 to the full version of this paper [18].

8. ACKNOWLEDGMENTS

We thank Faith Ellen and the PODC anonymous reviewers for many helpful comments on this paper. This work is partly supported by Singapore Ministry of Education Academic Research Fund Tier 2 grant MOE2011-T2-2-042, and partly supported by the research grant for the Human Sixth Sense Programme at the Advanced Digital Sciences Center from Singapore's Agency for Science, Technology and Research (A*STAR).

9. REFERENCES

[1] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. *Journal of Computer and System Sciences*, 73(3):245–264, May 2007.

[2] A. Blokhuis. On the sperner capacity of the cyclic triangle. *Journal of Algebraic Combinatorics*, 2(2):123–124, June 1993.

[3] A. R. Calderbank, P. Frankl, R. L. Graham, W.-C. W. Li, and L. A. Shepp. The sperner capacity of linear and nonlinear codes for the cyclic triangle. *Journal of Algebraic Combinatorics*, 2(1):31–48, March 1993.

[4] B. Chen, H. Yu, Y. Zhao, and P. B. Gibbons. The Cost of Fault Tolerance in Multi-Party Communication Complexity. *Journal of the ACM*, 2014.

[5] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *ICDE*, March 2004.

[6] G. Frederickson. Tradeoffs for selection in distributed networks. In *PODC*, 1983.

[7] R. Impagliazzo and R. Williams. Communication complexity with synchronized clocks. In *CCC*, June 2010.

[8] D. Kempe, A. Dobra, and J. Gehrke. Gossip-based computation of aggregate information. In *FOCS*, October 2003.

[9] F. Kuhn, T. Locher, and R. Wattenhofer. Tight bounds for distributed selection. In *SPAA*, 2007.

[10] F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic graphs. In *STOC*, 2010.

[11] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1996.

[12] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, December 2002.

[13] D. Mosk-Aoyama and D. Shah. Computing separable functions via gossip. In *PODC*, July 2006.

[14] S. Nath, P. B. Gibbons, S. Seshany, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. *ACM Transactions on Sensor Networks*, 4(2), March 2008.

[15] I. Newman. Private vs. common random bits in communication complexity. *Information Processing Letters*, 39(2):67–71, July 1991.

[16] B. Patt-Shamir. A note on efficient aggregate queries in sensor networks. In *PODC*, 2004.

[17] L. Shrira, N. Francez, and M. Rodeh. Distributed k-selection: From a sequential to a distributed algorithm. In *PODC*, 1983.

[18] Y. Zhao, H. Yu, and B. Chen. Near-optimal communication-time tradeoff in fault-tolerant computation of aggregate functions. Technical report, School of Computing, National University of Singapore, 2014. Also available at `http://www.comp.nus.edu.sg/~yuhf/optimalagg-nustr.pdf`.

Algorithm 2 The AGG Protocol. Following are some additional comments on the pseudo-code. By default, the sender of a message always attaches its id on the message (not shown in the pseudo-code), allowing the receiver to infer the sender. A "__" field in a received message means that we do not care about the value of that field. The pseudo-code allows a node to send multiple messages in a single round. In actual implementation, all these messages should be combined into one, and can thus be sent in one round. The pseudo-code invokes the **flood** primitive in several places, whose (trivial) implementation is not included in the pseudo-code. For a node to flood a message, the node sends the message to its neighbors. Any node receiving a flooded message simply forwards that message upon first receiving that message. The initiating node is called the *source* of the flooding. Note that if a node receives a second flooded message (potentially initiated by a different source) with the same content, the node will *not* forward it again. Finally, each node in AGG keeps track of the total number of bits it has sent. Once the number reaches $(11t + 14)(\log N + 5)$, a node will flood a special symbol to cause all nodes to abort AGG. This mechanism is not shown in the pseudo-code, for clarity.

```
1: /* Tree Construction Phase (total $2cd + 1$ rounds) */
2: if (I am the root) then
3:     $level = 0$; $parent =$ null; $children = \emptyset$; $ancestor[i] =$ null for all $i \in [1, 2t]$;
4:     send $\langle$tree_construct, $level$, $ancestor\rangle$ in round 1 of this phase;
5: else
6:     wait to receive the first message (with arbitrary tie breaking if multiple messages received in the same round) in the form of
       $\langle$tree_construct, $sender\_level$, $sender\_ancestor\rangle$ from any node $u$;
       // the node is now activated, and knows that the current round is round $sender\_level + 2$ of this phase;
       // the node can then determine the starting round of all the remaining phases in AGG and VERI;
7:     let $r = sender\_level + 2$ be the current round of this phase;
8:     $level = sender\_level + 1$; $parent = u$; $children = \emptyset$;
9:     $ancestor[1] = parent$; $ancestor[i] = sender\_ancestor[i-1]$ for all $i \in [2, 2t]$;
10:    send $\langle$ack, $parent\rangle$ in round $r$ of this phase;
11:    send $\langle$tree_construct, $level$, $ancestor\rangle$ in round $r + 1$ of this phase;
12: end if
13: upon receiving message in the form of $\langle$ack, $my\_id\rangle$ from any node $v$: $children = children \bigcup \{v\}$;

14: /* Aggregation Phase (total $2cd + 1$ rounds) */
15: $psum = my\_input$; $max\_level = level$; // $psum$ is for "partial sum"
16: for all $v \in children$ do
17:     if (in round $cd - level + 1$ of this phase, receive message $\langle$aggregation, $sender\_psum$, $sender\_max\_level\rangle$ from node $v$) then
18:         $psum = psum + sender\_psum$; $max\_level = \max(max\_level, sender\_max\_level)$;
19:     else
20:         flood $\langle$critical_failure, $v\rangle$ in round $cd - level + 1$ of this phase;
21:     end if
22: end for
23: send $\langle$aggregation, $psum$, $max\_level\rangle$ in round $cd - level + 1$ of this phase;

24: /* Speculative Flooding Phase (total $2cd + 1$ rounds) */
25: if (I am the root) then flood $\langle$flooded_psum, $my\_id$, $psum\rangle$ in round 1 of this phase;
26: if (I am not the root and no message from $parent$ is received in round $level + 1$ of this phase) then
27:     flood $\langle$flooded_psum, $my\_id$, $psum\rangle$ in round $level + 1$ of this phase;
28: end if

29: /* Partial Sum Selection Phase (total $cd + 1$ rounds) */
30: $ancestor[0] = my\_id$;
31: for all message received in the form of $\langle$flooded_psum, $source\_id$, __$\rangle$ do
32:     let $i \in [0, 2t]$ be the smallest $i$ such that $ancestor[i] = source\_id$; let $i = \infty$ if such $i$ does not exist;
33:     let $j \in [0, 2t]$ be the smallest $j$ such that $ancestor[j]$ is the root or $\langle$critical_failure, $ancestor[j]\rangle$ has been received;
        let $j = \infty$ if such $j$ does not exist;
34:     $dom$ = I have received a message $\langle$flooded_psum, $ancestor[k]$, __$\rangle$ with $k \in [i + 1, j]$; // $dom$ is for "dominated"
35:     if ($i \le t$) and ($i \le j$) then // I am a witness
36:         if ($j = \infty$) then flood $\langle$dominated, $source\_id\rangle$ in round 1 of this phase;
37:         if ($j \neq \infty$ and $dom$) then flood $\langle$dominated, $source\_id\rangle$ in round 1 of this phase;
38:         if ($j \neq \infty$ and (!$dom$)) then flood $\langle$compulsory||optional, $source\_id\rangle$ in round 1 of this phase;
39:     end if
40: end for

41: /* Output Phase (only executed by the root) */
42: $sum = 0$;
43: for all received message in the form of $\langle$flooded_psum, $source\_id$, $source\_psum\rangle$ do
44:     if ($\langle$compulsory||optional, $source\_id\rangle$ has been received) then $sum = sum + source\_psum$;
        // messages $\langle$dominated, $source\_id\rangle$ are not actually needed, and we sent those only for clarity
45: end for
46: output $sum$;
```

Algorithm 3 The VERI Protocol. The initial values of the variables $parent$, $children$, $ancestor$, $level$, and max_level are all from the previous AGG execution. All the comments in the caption of Algorithm 2 apply to Algorithm 3 as well, except the following: In VERI, once a node has sent $(5t + 7)(10 + 3 \log N)$ bits, it will flood a special symbol to terminate VERI and cause the root to output `false`.

```
/* Failed Parent Detection Phase (total $2cd + 1$ rounds) */
if (I am the root) then
    flood ⟨detect_failed_parent⟩ in round 1 of this phase;
else
    if (no message from $parent$ is received in round $level + 1$ of this phase) then
        flood ⟨failed_parent, $parent$, $max\_level - level + 1$⟩ in round $level + 1$ of this phase;
    end if
end if

/* Failed Child Detection Phase (total $2cd + 1$ rounds) */
if ($children = \emptyset$) then // I am a leaf
    flood ⟨detect_failed_child⟩ in round $cd - level + 1$ of this phase;
else
    for all node $v \in children$ do
        if (no message from node $v$ is received in round $cd - level + 1$ of this phase) then
            flood ⟨failed_child, $v$⟩ in round $cd - level + 1$ of this phase;
        end if
    end for
end if

/* LFC Detection Phase (total $cd + 1$ rounds) */
for all received messages in the form of ⟨failed_parent, $v$, __⟩ do
    let $i \in [0, 2t]$ be the smallest $i$ such that $ancestor[i] = v$; let $i = \infty$ if such $i$ does not exist;
    let $j \in [0, 2t]$ be the smallest $j$ such that $ancestor[j]$ is either the root or ⟨critical_failure, $ancestor[j]$⟩ was previously received in
    AGG; let $j = \infty$ if such $j$ does not exist;
    if ($i \leq t$ and $i \leq j$) then // I am a witness
        let $k \in [i, 2t]$ be the smallest $k$ such that i) ⟨failed_child, $ancestor[k]$⟩ has been received, or ii) $ancestor[k]$ is the root, or
        iii) ⟨critical_failure, $ancestor[k]$⟩ was previously received in AGG; let $k = \infty$ if such $k$ does not exist;
        if ($k - i + 1 \geq t$) then
            flood ⟨LFC_tail, $v$⟩ in round 1 of this phase;
        else
            flood ⟨not_LFC_tail, $v$⟩ in round 1 of this phase;
        end if
    end if
end for

/* Output Phase (only executed by the root) */
if (I have received message ⟨LFC_tail, $v$⟩ for any node $v$) then output false; // LFC exists
for all received message in the form of ⟨failed_parent, $v$, $x$⟩ where $x \geq t$ do
    if (⟨not_LFC_tail, $v$⟩ has not been received) then output false; // LFC may exist — VERI may have one-sided error here
end for
output true; // no LFC
```