LOT: Fast, Efficient and Robust In-Network Computation

André Allavena[*] and Srinivasan Keshav
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{aallaven,keshav}@uwaterloo.ca

Technical Report CS-2006-22

Last modified July 18, 2006 – 23:16

10 + 4 pages = 14 pages

Keywords: robust, duplicate-insensitive, in-network computation, fault-tolerant, gossip/epidemic algorithms, tree abstraction.

Abstract

Today, companies such as eBay, Amazon, Google, and IBM routinely operate clusters with more than 10,000 servers located in data centres around the world. Developing applications that efficiently use the resources of such a distributed cluster is challenging. This task is made eased by a middleware layer that provides application programmers with the illusion of dynamically updated "global state". Maintaining global state can be viewed as repeatedly computing an arbitrary function over a set of time-varying values, where the values are held at each node, and every node needs to know the resultant function. Many well-known problems in distributed systems, such as load balancing, leader election, barrier synchronisation, distributed file system maintenance, and coordinated intrusion detection can be cast in this form.

We present and rigorously analyse an algorithm that uses a a tree of virtual nodes to compute nearly arbitrary functions of global state. Our scheme is fast: running time is $\Theta(\ln n)$. It is efficient: nodes exchange $\mathcal{O}(n \ln n)$ messages. Most of these messages are within a data centre and therefore are relatively cheap. It is accurate: the computed value does not have inherent errors either due to double-counting, as with standard gossip, or due to stochastic counting, as in the Flajolet-Martin approach. Finally, it is fault tolerant: The algorithm fails with probability $\mathcal{O}\left(\frac{1}{n^{c(1+\rho)}}\right)$ where c is a constant and $0 \leq \rho \leq (\ln n)^{Cst}$ a user-chosen reliability parameter. We therefore believe that our work is the basis for building robust and efficient distributed systems in a variety of problem domains.

[*]alternate email address: andre@cs.cornell.edu

1 Background and Related Work

Rapidly dropping hardware prices and the proliferation of data centres has made it possible to build multi-data centre clusters with thousands or even tens of thousands of machines managed by a single administrative entity. Today, companies such as eBay, Amazon, Google, and IBM routinely operate clusters with more than 10,000 servers each. Such clusters have three important properties:

- All servers, which we call *nodes* in this paper, are roughly equal in their processing capabilities.
- Every server can talk to every other server using an underlying network.
- Server failure is common: at any given time, up to 5 % of the servers may be unavailable.

Developing applications that efficiently use the resources of a distributed cluster is challenging. This task is eased by a middleware layer that provides application programmers with the illusion of dynamically updated "global state". Examples of such state are: the set of servers that currently provide a particular service; the load at each server; and the "scoreboard" set of servers that have completed a given task. In general, maintaining global state can be viewed as repeatedly computing an arbitrary function over a set of time-varying values, where the values are held at each node, and every node needs to know the resultant function. Many well-known problems in distributed systems, such as load balancing, leader election, barrier synchronisation, distributed file system maintenance, and coordinated intrusion detection can be cast in this form [8].

We now make the problem more precise. Consider a distributed system that has $n(t)$ live nodes at time t (node failures and additions may change this set over time). Let $v(t)$ be a *global state vector* of time-varying values $v_f(t)$ held at the nodes indexed by f, and let $g(v(t))$ be a function defined over this vector. The goal of our work is to define a distributed computation initiated at time t_0 such that, at the end of the computation, every live node in the system knows $g(v(t_0)$.

Past work in this area falls into three categories. The first approach induces a tree on the set of nodes and computation proceeds along the tree from leaves towards the root. Trees are efficient, requiring only $\mathcal{O}(\ln n)$ time steps, and only $\mathcal{O}(n)$ messages, the least necessary. Unfortunately, tree topologies are inherently brittle because the failure of a single node can disrupt the computation. Therefore, the research focus has been to make the tree computation more robust. This is also the approach taken in our work.

The work most closely related to ours is Astrolabe [13]. In this system, an administrator-managed tree hierarchy uses a complex monitoring and leader election process at each level of the tree to make it robust. Despite this, Astrolabe is unable to tolerate even moderate failure rates, because a double failure in a single Astrolabe zone can cause temporary tree partitioning. Moreover, the system only makes weak consistency guarantees.

An alternative approach, also in this category, is to use k trees in parallel, with each node value replicated k times, hoping that at least one of the tree computation succeeds. However, this approach fails with even k well-chosen (or simply unlucky) failures because a single failure in each tree partitions it.

The second line of work is to use a gossip-based computation. Epidemic (or gossip) algorithms are known to be the most robust and efficient way to disseminate information in a network, and recent work has used a gossip-based approach for in-network computation as well. In a gossip, at each round of computation, a node randomly chooses some other node and exchanges a partially evaluated function value with it. The aggregate of the partial values is then gossipped in the next computation round. This results in a high degree of fault-tolerance. Essentially, once a node has gossipped its value to a few other nodes, its subsequent failure does not affect the final result. Moreover, the computation completes in $\mathcal{O}(\ln n)$ rounds.

The main problem with gossip-style approaches is double-counting, that is, the problem that some node value may be incorporated into the function value more than once. Tree-style computation does not have this problem because there are no cycles in the evaluation graph. We can avoid double counting in gossip systems by converting non-extremal functions (such as *sum*) to functions that only depend on extremal values (such as *max*). A popular approach uses Flajolet-Martin [5] stochastic counting, which converts *sum* to *max* [4, 11]. However, the intrinsic nature of the Flajolet-Martin algorithm (probabilistic computation of the logarithm) results in errors of up to 50 %. To increase accuracy, further work [9] uses this approach only in faulty regions of the network and uses an (accurate) spanning tree in the non-faulty regions. Reference [10] studies the running time for arbitrary graphs. Nevertheless, the Flajolet-Martin approach is not general, and only applies to a fairly narrow range of functions.

The third line of past work gets around the double counting problem, while still preserving accuracy, by associating *weights* with partial function values [7]. As long as the weights sum to unity, double counting is avoided. Reference [2] characterises the convergence speed of such algorithms. This style of computation has also been recently used by [6], which shows that the message overhead can be reduced to $\mathcal{O}(n \ln \ln n)$. Although this approach is tolerant to link failures, it is not tolerant to node failures during computation because the failure of a single node during the computation causes the sum of node weights to not sum to unity, perturbing the final result.

To sum up, existing solutions do not simultaneously provide accuracy and fault tolerance. In contrast, our approach, which essentially provides robust trees, is as accurate as a tree, but as fault-tolerant as a gossip.

Contributions We present a fast, efficient, accurate and fault-tolerant algorithm for computation of functions of global state.

- *Fast:* The running time of our scheme is $\Theta(\ln n)$.
- *Efficient:* Nodes exchange $\mathcal{O}(n \ln n)$ messages. Most of these messages are within a data centre and therefore are relatively cheap. The number of costly messages between data centres is, with high probability: $\mathcal{O}\left(n + n\frac{\ln \ln n}{1+\rho}\right)$, where $0 \leq \rho \leq (\ln n)^{Cst}$ is a reliability parameter. Let $\mathcal{A}gg$ denote the size of the partial result of the aggregation function g, typically $\mathcal{O}(1)$. Then, almost all messages are of size $\mathcal{O}(\mathcal{A}gg + m)$, where $m = \mathcal{O}((1+\rho)\ln n \ln \ln n)$. For functions that need to be distorted to be embedded in the tree, the additional cost is an increase in message size proportional to the distortion.
- *Accurate:* The computed value does not have inherent errors either due to double-counting, as with standard gossip, or due to stochastic counting, as in the Flajolet-Martin approach.
- *Fault-tolerant:* The algorithm fails with probability $\mathcal{O}\left(\frac{1}{n^{c(1+\rho)}}\right)$ where c is a constant.

Our scheme can also provide every node with several spatially-decaying aggregates [3], *i.e.*, evaluations of the function on the nodes within tree distance d, for varying d.

The paper is organised as follows: in Section 2 we present our algorithm which essentially performs computation on a virtual tree. In Section 3 we show how to map virtual nodes to physical nodes in the face of node failures and arrivals. Section 4 analyses the performance of our scheme, Section 6 concludes.

2 Approach

To fix ideas, consider a computation of g on a full binary tree. Each leaf f initially holds a value v_f. In the first *round*, the level 1 nodes, *i.e.* the nodes that are parents of the leaves, compute the partial function values $g(v_f, v_{f+1})$ for $f = 1, 2, 3, ..., n/2$. At the end of the first round, the level 1 nodes have computed $n/2$ partial values of g. In the second round, the level 2 nodes compute g from the values computed in the first round, *i.e.* compute the partial values $g(v_f, v_{f+1}, v_{f+2}, v_{f+3})$ $= g(g(v_f, v_{f+1}), g(v_{f+2}, v_{f+3}))$ for $f = 4j,\ j = 1, 2, 3..., n/4$.

For message sizes to grow sub-linearly, g must be *aggregable* in some fashion. That is, the representation of the partial values should grow sub-linearly with the number of arguments. This is trivially true for functions such as *max* and *sum*, but is false for a function like *list*, which lists its arguments.

2.1 Virtual Tree

We compute g on a *virtual tree* where each non-leaf node is a *virtual node*, emulated by some of the leaves. In the sequel, we refer to real (or physical) nodes as *leaves*, and to virtual nodes as *vnodes*. Each vnode has between b and $2b$ children, and we denote by h the height of the tree (Figure 1).

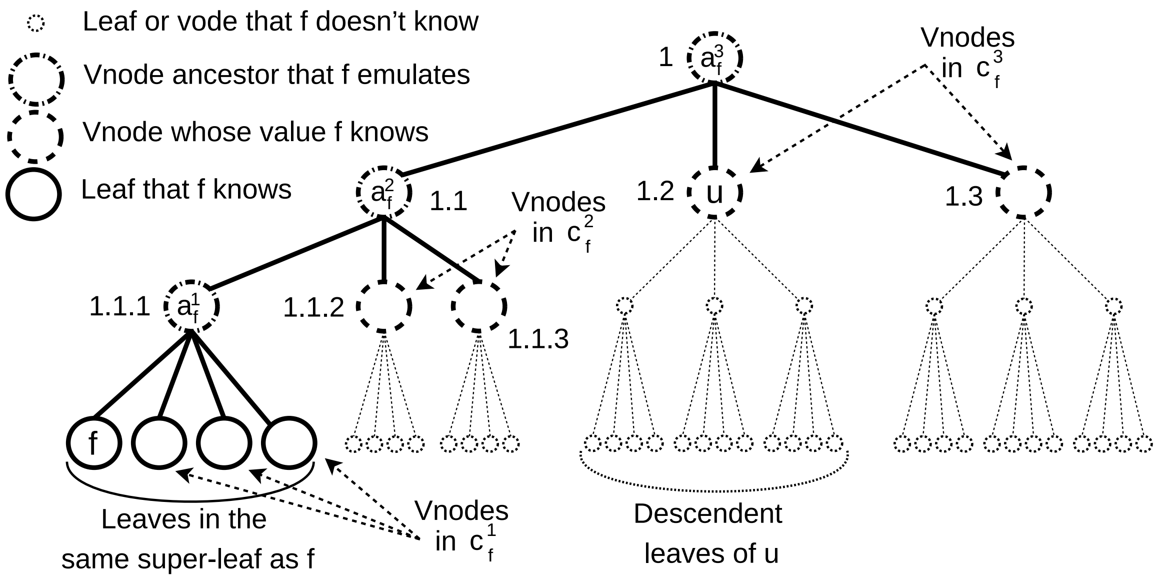

Figure 1: Virtual tree from the perspective of leaf f

Each leaf f is responsible for emulating all its (vnode) ancestors. Denote the i^{th} ancestor of f by a_f^i and the set of all ancestors by $\mathcal{A}_f$. For example, a_2^1 is the parent of leaf 2. Symmetrically, $\mathcal{D}_u$ is the set of descendent leaves of vnode u, that is, the set of leaves emulating u.

Define the i^{th} *contact set* of f as the set of vnodes $\mathcal{C}_f^i = \{c \neq a_f^{i-1}$: c is a child of $a_f^i\}$. In other words, $\mathcal{C}_f^i$ is the set of children of leaf f's i^{th} ancestor, excluding the $i-1^{th}$ancestor itself (these vnodes are the $i-1^{th}$ level). The contact set of f, denoted $\mathcal{C}_f$ is the union of these contact sets.

A leaf f has a *mapping table* $M_{\mathcal{C}_f}$ that maps every element $c \in \mathcal{C}_f$ to a set of m leaves (in $\mathcal{D}_c$) that emulate c. If c is a vnode at level i in the tree, then each leaf f can choose m leaves from between b^i and $(2b)^i$ possible choices to create its mapping table.

2.2 Algorithm

Leaf f computes the partial function value of g at level i, that is, the value at vnode a_f^i, by applying g over a_f^i's children. These child vnodes are the elements of $\mathcal{C}_f^i$ together with ancestor vnode a_f^{i-1}.

Assuming f already knows a_f^{i-1} from a previous round of computation, this amounts to using $M_{\mathcal{C}_f}$ to find some nodes that emulate each vnode in $\mathcal{C}_f^i$, contacting them to retrieve their partial function values, applying g to these values, then saving this value as the partial function value at a_f^i.

For example, in Figure 1, in the first round of computation, the leaf marked f uses M_f^1 to find its siblings, retrieves values at these siblings, and uses these to compute the value of vnode `1.1.1`. Other leaves in parallel compute the value of vnodes `1.1.2` and `1.1.3`. In the second round, leaf f uses $M_{\mathcal{C}_f^2}$ to contact one of the leaves in charge of emulating vnodes `1.1.2` and `1.1.3` and retrieves these vnode values. Leaf f can now apply g to these values to compute the value of vnode `1.1`. Meanwhile, other leaves compute the values of vnodes `1.2` and `1.3` in parallel. Proceeding in this fashion, after h rounds of computation, every leaf computes the value of g at the root, as desired.

So far, we assumed that all messages take a unit time. However, in a real system, some leaves will be slower than others. We deal with this by making a leaf block while waiting for a reply from another leaf. This allows leaves to make as much progress as is possible, given the underlying heterogeneity in the system.

Main Algorithm

```
Round 1: do INITIALISATION; GOSSIP 2
for Round $i = 2$ to $i = h$ do
    INITIALISATION
    ACQUISITION 1
    GOSSIP 1
    ACQUISITION 2
    GOSSIP 2
end for

INITIALISATION
do nothing

GOSSIP 1:
epidemic broadcast of the values

GOSSIP 2:
epidemic broadcast of the missing values and
consensus on who contacts whom in next round

ACQUISITION 1: (Round $i$)
for all vnodes $c_j$ in $\mathcal{C}_f^i$ do
    $k_1$ leaves try to contact a leaf emulating $c_j$
end for

ACQUISITION 2: (Round $i$)
for all $c_j$ whose value we are missing do
    $k_2$ leaves try to contact a leaf emulating $c_j$
end for
```

2.3 Super-leaves

The algorithm in Section 2.2 is suitable for systems where all leaves are "close" to each other, so that inter-leaf message communication costs are uniformly small. In reality, some links will cost less than others. Future massively parallel distributed systems are likely to span the globe, consisting of clusters of servers co-located in the same data centre. Communication within a cluster (*i.e.* within a data centre) would incur a delay of less than 1 ms. In contrast, communication between data centres can take tens or hundreds of milliseconds. To achieve good performance, this link heterogeneity has to be incorporated into our algorithm.

We do so by grouping between l and $2l$ "nearby" leaves into a *super-leaf.* Leaves in the same super-leaf are expected to be in the cluster or same data centre. Because they have very low communication costs, leaves in the super-leaf can use epidemic or gossip style algorithms to maintain a consistent view of each other's state. Therefore, in the sequel, we will assume that they use a standard group communication protocol to achieve group consensus. This group communication takes $\mathcal{O}(\ln l)$ time, with a small constant multiplier.

We choose $b = (\ln n)^{1/\lambda}$ and $l = (1+\rho) \cdot (\ln n)^{1+1/\lambda}$ where λ is a constant whose choice is left to the implementor and is typically 2 or 3. The parameter ρ characterises reliability and is such that $0 \leq \rho \leq \ln^c n$ where c is a constant. These values are necessary to derive the bounds of Section 4.

All leaves in a super-leaf share the same vnode parent, so they all have the same ancestor set $\mathcal{A}$ and the same contact set $\mathcal{C}$. Moreover, they use a consensus algorithm to agree on consistent values of M_{C_f} as described in Section 3.

As before, leaves in a super-leaf need to fetch values in $\mathcal{C}$ to compute g. However, instead of having *every* leaf retrieve *all* values in C, at each round super-leaves save communication costs by distributing the work of fetching values across leaves, then sharing the result.

The algorithm works in two phases. In the first phase of each round, leaves use a consensus algorithm to select $k_1 = \Theta(\ln \ln n)$ leaves (see Section 4 for the exact value of k_1) for each of the up to $2b - 1$ contacts whose values need to be acquired. These values are then broadcast to all leaves within the super-leaf in $\mathcal{O}(\ln l)$ time. Due to leaf failures, some values may not have been retrieved. If so, in the second phase $k_2 = \frac{(1+\rho)}{2} \ln n \approx \ln n$ leaves are assigned to fetch values from each of the missing contacts. These values are then broadcast to all the leaves.

Due to randomness in M across different leaves, some leaves will get more value retrieval requests than others. Because request handling is serialised, an overloaded leaf could potentially slow down the entire system. To avoid this, we define a parameter `max_load` $= \ln \ln n$. If a leaf receives more then `max_load` requests per phase, it drops all excess requests. We show in Section 4 that this guarantees that no leaf is overloaded, yet, with high probability, the function computation successfully completes because all super-leaves are able to get all necessary values.

3 Creating and Maintaining Mapping Tables

Define set $M = \bigcup_f M_{C_f}$ to be the aggregate of all the leaf mapping tables. M should satisfy two properties:

- Every leaf should appear in M an equal number of times: leaves that appear more frequently than others are likely to get proportionately more requests, and the resulting load imbalance can slow down the overall computation.
- Failed leaves should be weeded out from M.

Our solution meets these goals by providing every leaf that needs to contact vnode u with a uniform and mostly uncorrelated random sample of live leaves in $\mathcal{D}_u$. This is done by piggybacking a sample of representative leaves emulating vnode u when returning the partial function value at u. These samples are used to update M, and the updated version of M is used in the *next run* of the distributed computation. Only those leaves that are alive at the beginning of the run at time t will be included in the mapping tables used by the algorithm at run $t + 1$. Hence, dead leaves are removed at the end of every invocation of the distributed computation.

The mapping from vnodes to a set of representative leaves using the mapping table M is similar to an IP anycast address being mapped to a set of representatives [1]. Indeed, we could have used IP anycast addresses for vnodes, and made the descendent leaves members of the appropriate sets of anycast groups. However, existing anycast approaches do not, to our knowledge, have the fault tolerance and load balancing properties of our approach. Similarly, Astrolabe [13] also requires a mapping from a “zone” to its representatives. The solution implemented in this system, however, differs greatly from ours.

3.1 Algorithm

We now describe how a leaf f creates and maintains its mapping table $M_{\mathcal{C}_f}$. Recall that computation proceeds in rounds indexed by i, $1 \leq i \leq h$, where, during round i, the i^{th} level vnode values are computed in parallel. In addition, during round i, the mapping tables $M_{\mathcal{C}_f^i}$ of the next run are prepared.

Each leaf f holds two size-m vectors, $\texttt{sample}_f^i$ and $\texttt{temp}_f^i$. At the beginning of round i, both vectors are set to $\texttt{temp}_f^{i-1}$, a near uniform random sample of $\mathcal{D}_{a_f^{i-1}}$ (where a_f^{i-1} is f's height $i-1$ ancestor). When f is contacted during round i by some leaf x and asked about the partial function value at a_f^{i-1}, leaf f also piggybacks k_2 distinct, not previously returned, elements from $\texttt{sample}_f^i$ in its reply (unless there has been a structural change in the tree, which is discussed in Section 3.3). We choose $m = 2k_2 \cdot \texttt{max\_load}$ so that a leaf can always return a new set of k_2 elements. In addition, leaf f and leaf x swap content of their $\texttt{temp}^i$ vectors.

Then, during the gossip phases, the leaves that retrieved the values of some vnodes c_j gossip the values $v(c_j)$ along with the new representative samples of c_j. For each c_j, the samples (if more than 1) are randomly merged to update the mapping table entry corresponding to c_j, which will be used by the super-leaf during the next algorithm run. In addition, during each iteration of the epidemics with a leaf y, leaf f and y swap a randomly chosen half of their $\texttt{temp}^i$ elements. At the end of the epidemics phase, $\texttt{temp}_f^i$ vector is a near uniform random sample of m elements from $\mathcal{D}_{a_f^{i+1}}$ where a_f^{i+1} is f's height $i+1$ parent.

Additions to the main algorithm: Incoming request at leaf f (outgoing is symmetric)

```
*INITIALISATION*
Round 1:
  $\texttt{temp}_f^1 :=$ $m$ copies of $f$'s address
Round $i$: $(1 < i \leq h)$
  $\texttt{temp}_f^i := \texttt{temp}_f^{i-1}$
  $\texttt{sample}_f^i := \texttt{temp}_f^{i-1}$
*GOSSIP (1 & 2)*
**for** each incoming connection, from leaf $y$ **do**
  swap a random half of $\texttt{temp}_f^i$ and $\texttt{temp}_y^i$
**end for**
*ACQUISITION (1 & 2)*
**for** each incoming connection, from leaf $x$ **do**
  **if** received more than max_load connections
  in total **then**
    drop the connection and exit
  **end if**
  swap $\texttt{temp}_f^i$ and $\texttt{temp}_x^i$
  **if** tree structure($a_{i-1}^f$) changed **then**
    return appropriate $k_2$ sample(s)
  **else**
    return $k_2$ new elements from sample
  **end if**
**end for**
(gossip and acquisition can be intertwined)
```

Consider a vnode u of height i. The concatenation Cat of the $\texttt{temp}_f^i$ vectors over all leaves $f \in \mathcal{D}_u$ contain exactly m copies of each leaf. At the end of round i of run t, Cat is a random permutation (though correlated to Cat of run $t-1$). During round $i+1$, each super-leaf that contacts u is provided with a distinct, non-overlapping portion of Cat, which will be the mapping used to contact u during round $i+1$ of run $t+1$. The algorithm attempts to maximise mixing of Cat, and alternatives are certainly possible. Our simulations (not included) suggest that the mixing is sufficient. Note that the algorithm we describe later to handle tree structural changes only approximatively maintains the non-overlapping property; this is the subject future work.

3.2 Dealing with Leaf Joins

For now, assume that members joining or leaving the system do not break the constraints on b and l, thus do not warrant a change in the tree structure. Structural changes are described next.

A gracefully departing leaf initialises its `temp` vector to nil instead of m copies of itself at the beginning of its last run. A leaf crashing after the start of the t^{th} run of the algorithm but before the start of the $t+1^{th}$ run is expunged from the system by the time run $t+2$ starts.

A leaf f joins a super-leaf by boot-strapping from any leaf y of the super-leaf, once the current run is over. Leaf f participates in the next run like any other leaf, initialising its vector `temp` to m copies of itself. Non-local leaves will not contact f during its first run. However, leaf f will be contacted like any other leaf during subsequent runs. While f could join any super-leaf, for performance reasons, it is important that f gets included in the super-leaf of nearby members, since most of the communications are within the super-leaf.

This can easily be done in the case of data centres and operators who know the rack disposition of machines. Otherwise, the joining member may do a descent on the virtual tree and converge to the closest super-leaf, assuming an appropriate metric space. Alternatively, system members could participate in a system like Meridian [14] and use Meridian to locate an optimal super-leaf to join.

3.3 Structural Changes

The tree is maintained in the same way as a standard B-trees. When a super-leaf goes over the size limit of $2l$, it splits. When it becomes smaller than minimum size of l, it merges with a nearby super-leaf. (Alternatively, it can dissolve and each component leaf can individually rejoin the system.) Then, structural changes are propagated up the tree, if needed, to maintain the vnode branching between $b =$ and $2b$. Recall $b = (\ln n)^{1/\lambda}$ (and l) are not constant.

Note that even though the tree is height-balanced, the number of super-leaves in subtrees of the same height may vary significantly. Define the *weight* of vnode u as the number of super-leaves descending from u. The algorithm can tolerate a moderate weight imbalance, however a significant imbalance will impact the load, or equivalently, the running time. In addition to typical B-tree updates, our tree is also re-balanced to limit weight imbalances, as described below. To do so, the algorithm also keeps track of the weight of each vnode.

Consider virtual node u of height $i-1$. During round i, some leaves (not in $\mathcal{D}_u$) try to learn u's value. Denote by $\mathrm{acq}(u)$ the number of these leaves and let $\alpha_u = \frac{\mathrm{acq}(u)}{|\mathcal{D}_u|}$. Variable α_u is the expected load on a leaf during round i. A re-balancing is triggered when ($\eta \geq 0$ is user chosen):

$$\alpha_u > \alpha_{\max} = \texttt{max\_load} \cdot e^{-1-(2+\eta)\frac{\ln \texttt{max\_load}}{\texttt{max\_load}}} \approx \frac{\texttt{max\_load}}{e} \qquad (1)$$

denoting that vnode u has insufficient weight. To address this, we transfer some branches from the heaviest sibling(s) of u to u. See Section 4 for a justification of the value of the maximum expected load $\alpha_{\max}$ and upper bounds on $\mathrm{acq}(u)$. Because the bound is independent of the subtree size, re-balancing can only be required for higher tree levels, where changes themselves are infrequent.

Structural changes are carried in control messages appended to regular messages. Membership changes and the split and merge structural changes they trigger, if any, are carried out at the same time. Round i of run t provides leaves with the mapping information used by round i of run $t+1$, including the number of subtrees rooted at their height-i ancestor v. At the end of round i of run t, each leaf f checks whether the tree constraints on a_f^i are satisfied by the new mapping. If not, any such leaf f selects the appropriate change (a_f^i splitting into two or a_f^i merging with some a_x^i) and makes the appropriate change to its mapping table for use in the next algorithm run. Leaf f also incorporates the change in its announcement for round $i+1$ of the current run.

At the end of round i, leaf f also checks whether there is a weight imbalance between the level i vnodes, that is, between a_f^i and the c_f^{i+1}. If so, re-balancing happens: some branches are transfered (or received) by a_f^i from an appropriate c_f^{i+1}. Note that unfortunately, the weight of these vnodes is stale information: the updated c_f^{i+1} weights are only received during round $i+1$. In other words, the membership changes and the split / merges they trigger are enacted at the same time. However, the re-balancing these may trigger is taken into account a run later.

The choices of what and how to split, merge ore re-balance are deterministic so that the same decision is reached by all super-leaves independently, with no communication. During round $i+1$, leaf f (f is queried about a_f^i) will announce the changes it made to a_f^i and provide the necessary membership information to the leaves contacting it. If the structural change is a split, leaf f returns two size-m random sample of elements, containing elements of each side of the split. In case of a merge, if the query comes from a leaf $x \in \mathcal{D}_v$, leaf f returns a size-m random sample of $\mathcal{D}_u$ for each u child of v; if the query comes from a leaf $x \notin \mathcal{D}_v$, leaf f returns no sample. A re-balancing is similar to a merge; however, only samples for the subtrees being transfered are returned. Finally, if there is no structural change, leaf f returns a random sample of leaves emulating v.

4 Analysis

4.1 Running Time

Theorem 1. *The running time of the algorithm is $\Theta(\ln n)$.*

Proof. The running time is $\mathcal{O}(h \cdot R)$ where R is the running time of a round. We have $\log_{2b} \frac{n}{2l} + 1 \le h \le \log_b \frac{n}{l} + 1$, or $h = \Theta\left(\frac{1}{\ln b} \ln \frac{n}{l}\right)$. There are two operations during a round: the acquisition of values and the broadcast within the super-leaf, each of which may be repeated once more if some values are missing. The running time of the broadcast within each super-leaf is $\Theta(\ln l)$ when running an epidemic algorithm. Because each leaf sends one or two messages and processes at most 2 `max_load` incoming messages the length of both acquisition phases is upper bounded by $2\,\texttt{max\_load} + 2 = 2 \ln \ln n + 2 \le 2 \ln l + 2$. We have $R = \Theta(\ln l + \texttt{max\_load}) = \Theta(\ln \ln n) = \Theta(\ln b)$. The running time is then

$$T = h \cdot R = \Theta\left(\frac{1}{\ln b} \ln \frac{n}{l} \cdot \ln b\right) = \Theta(\ln n) \qquad \square$$

4.2 Reliability in the Face of Failures

We now prove that the algorithm succeeds with high probability even in the presence of failures. This derivation also justifies our earlier choices of values for l and b.

4.2.1 Model and Assumption

We assume that a fraction p of leaves, chosen uniformly at random amongst the members currently in the membership list, may fail at anytime during the computation (as opposed to before the start of the computation, as prior work [7, 6] does). We assume the failures to be fail-stop failures. The value of a failed leaf will be included in the computation if the value makes it to a leaf that will stay alive during the computation. Typically, the value is dropped only if it fails to be forwarded to a super-leaf member that stays alive until (at least) the completion of round 1. Leaves stopping during the execution of the algorithm do not disrupt it. This handling of failed leaves is similar to that of prior work [4, 11, 10, 9] that handle members failing during computation.

We assume that the mapping tables provide each leaf with a uniform random sample of leaves that it needs to contact, although in practice the mapping is only approximatively uniform.

Assuming that leaves select uniformly at random which connections to drop when they become too numerous, we model the dropping of incoming connections by an increase bounded by ε in the probability of failures from p. The justification for this and the new modified probability of failure $p'$are described below. The detailed proofs of the theorems can be found in appendix.

4.2.2 Reliability

Recall that $\alpha_u = \frac{\mathrm{acq}(u)}{|\mathcal{D}_u|}$ is the expected load on the leaves of $\mathcal{D}_u$ and that the tree re-balancing ensures that, for a user-chosen $\eta \geq 0$ (see the bound on ε below):

$$\alpha_u \leq \alpha_{\max} = \texttt{max\_load} \cdot e^{-1-(2+\eta)\frac{\ln \texttt{max\_load}}{\texttt{max\_load}}} \tag{1}$$

Lemma 1. *Leaves dropping connections when their load exceeds* `max_load` *long range messages in a round is equivalent to leaves serving all their connections but with an increase of ε in the failure probability of targets, where $\varepsilon \leq \frac{1}{4\ln^2 \mathit{max\_load}} \cdot \frac{1}{(\mathit{max\_load})^\eta}$.*

We set $k_1 = \frac{\ln\ln n}{|\ln(2p+\varepsilon)|}$. Note that $\lim_{n\to\infty} \varepsilon = 0$.

Theorem 2. *The algorithm fails with probability:*

$$p_{fail} = \mathcal{O}\left(\frac{1}{n^{(\gamma-1)+\gamma\rho}}\right)$$

where $\gamma = |\ln(2(p+\varepsilon))|/2$ is a parameter depending on the failure probability, and ρ a user-chosen parameter governing the reliability with $0 \leq \rho \leq \mathcal{O}(\ln n)$ being acceptable values. Larger ρ leads to larger super-leaves, of size $l = (1+\rho)\cdot(\ln n)^{1+1/\lambda}$

4.3 Load on Leaves and Tree Balance

The membership service needs to maintain the bound on α as defined in Equation 1. Denote by $S(u)$ the number of super-leaves that need to acquire u's value. Note that $\mathrm{acq}(u) \leq S(u)\cdot 2l/b = 2(1+\rho)S(u)\ln n$ since each super-leaf sends at most $2l/b$ messages to $\mathcal{D}_u$. For the highest vnodes in the tree, the bound on $\mathrm{acq}(u)$ is loose by a $(1+\rho)\ln n$ factor.

Theorem 3. *For the virtual nodes u high in the tree, that is, such that*

$$S(u) \geq (1+\rho)|\ln(2(p+\varepsilon))|\frac{\ln^2 n}{2e^2} \tag{2}$$

we have $acq(u) = \mathcal{O}(S(u)\cdot k_1) = \mathcal{O}(S(u)\cdot \ln\ln n)$

This result holds with very high probability. The probability that some (virtual) node exceeds the bound during the execution of the algorithm is smaller than the algorithm failure probability.

In other words, none of the vnodes satisfying constraint (2) will have a load exceeding $\mathcal{O}(k_1 {\cdot} S(u))$

4.4 Message Complexity

The number of messages sent by the algorithm is $\mathcal{O}(n\ln n)$. However, not all messages have the same cost. A more interesting metric is to count the number of long distance messages (as opposed to local messages, sent within the super-leaf).

Theorem 4. *The number of non-local messages sent by the algorithm is with high probability $\mathcal{O}(n)$.*

Denote by $\mathcal{A}gg$ the size of the partial result of the aggregation function g, typically $\mathcal{O}(1)$.

Theorem 5. *Non-local messages, excluding merging and re-balancing ones, are of size $\mathcal{O}(\mathcal{A}gg+m)$. Tree merging and re-balancing messages are of size $\mathcal{O}(\mathcal{A}gg + l)$*

The size of local messages is implementation specific, typically of size $\mathcal{O}(l)$.

5 Computable Functions

We call the "dependency graph" of a function the directed acyclic graph representing the dependency of intermediary computations on other intermediary results and variables. Variables are sink nodes; the other nodes represent operands. A node's out-going edges point to the operand inputs.

Any function whose dependency graph can be naturally embedded into the tree the algorithm utilises can be computed efficiently using our scheme, such as:

1. Sum, product, mean, standard deviation and other moments;
2. Count of nodes having a particular property, which includes voting;
3. Min, max and derived expressions: top k, consensus, barrier synchronisation (by waiting for the minimum value to increase) [13],
4. Approximate histogram [12] and therefore approximate cumulative distribution and most frequent values;
5. Uniform and non-uniform random sampling.

Theorem 6. *All dependency graphs can be embedded in the algorithm tree, therefore all functions are computable.*

This comes at the expense of larger messages. See details in full paper. The idea is to propagate up the tree all values that cannot be processed. In the worst case, no aggregation is done, yielding $\mathcal{O}(n)$ sized messages. Randomisation is handled by reaching agreement on a generator seed.

6 Conclusion

We have described the first accurate, efficient and robust algorithm for in-network computation of functions of global state. We provide a simple abstraction of a tree on which arbitrary functions can be evaluated. Our scheme has three important features:

- Unlike other systems such as peer-to-peer overlays, our solution updates routing tables during function computation. By deriving the tables for the next run from scratch during the current one, our scheme deals well with node churn.
- We exploit the fact that network link latencies are heterogeneous. Our scheme groups nearby nodes into "super-leaves" and most communications are within the super-leaf. Other overlay schemes ignore this heterogeneity and consequently suffer from poor performance.
- Unlike other schemes, our routing tables contain only *representative samples* of virtual nodes and not exact matches; this flexibility inherently increases fault tolerance.

We believe that our work can serve as a basis for building robust and efficient distributed systems in a variety of domains.

References

[1] H. Ballani and P. Francis. Towards a global IP anycast service. In *Proc. SIGCOMM*, 2005.

[2] S. Boyd, A. Ghosh, B. Prabhakar, and D. Shah. Gossip algorithms: Design, analysis and applications. In *Proc. 24th IEEE INFOCOM*, 2005.

[3] E. Cohen and H. Kaplan. Spatially-decaying aggregation over a network: model and algorithms. In *Proc. 23rd ACM SIGMOD*, 2004.

[4] J. Considine, F. Li, G. Kollios, and J. Byers. Approximate aggregation techniques for sensor databases. In *Proc. 20th ICDE*, March 2004.

[5] P. Flajolet and N. Martin. Probabilistic counting algorithms for data base applications. *JCSS*, 31:182–209, 1985.

[6] S. Kashyap, S. Deb, K.V.M. Naidu, R. Rastogi, and A. Srinivasan. Efficient gossip-based aggregate computation. In *Proc. 25th ACM PODS*, 2006.

[7] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *Proc. 44th IEEE FOCS*, 2003.

[8] S. Keshav. Efficient and decentralized computation of approximate global state. *ACM Computer Communication Review*, January 2005.

[9] A. Manjhi, S. Nath, and P. Gibbons. Tributaries and deltas: Efficient and robust aggregation in sensor network streams. In *Proc. 24th ACM SIGMOD*, 2005.

[10] D. Mosk-Aoyama and D. Shah. Computing separable functions via gossip. In *Proc. 25th ACM PODC*, 2006.

[11] S. Nath, P. Gibbons, S. Seshan, and Z. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. 3rd ACM SENSYS*, 2004.

[12] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proc. 3rd ACM SENSYS*, 2004.

[13] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 21(2), May 2003.

[14] B. Wong, A. Slivkins, and G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proc. SIGCOMM*, 2005.

A Proofs of Section 4

Proof of lemma 1. First note that the leaves whose connections are dropped are chosen uniformly at random:

Let H be a descendent leaf of a virtual node u. Consider the set of leaves that are set to learn u's value during the round and assume that H will drop some connections. Break the process of leaves selecting their communication partner uniformly at random into two steps. First, an oracle chooses the histogram distribution of the load of the potential partners and randomly attaches a leaf to each load. (The probability distribution of the histograms is the same as if leaves selected their targets uniformly at random.) Then, each leaf selects at random the load of its communication partner – the probability is proportional to the load – and attempts to communicate with the leaf responsible for that load. This process is the same as if leaves chose their communication partner uniformly at random

Thus it is equivalent to consider that leaf targets fail uniformly at random with probability $p' = p + \varepsilon$ where ε is the probability that a connection is dropped. We now compute an upper-bound on ε.

Consider leaf f, descendent of virtual node u. Denote by $s = |\mathcal{D}_u|$ the number of leaves that emulate u and denote by m the number of leaves that try to learn u's value during the current round. By our assumption, each one of these m leaves selects a leaf uniformly at random in $\mathcal{D}_u$. The probability that leaf f has received j connections is therefore $\binom{m}{j}\left(\frac{1}{s}\right)^j\left(1-\frac{1}{s}\right)^{m-j}$. Denote by E the expected number of connections that f drops. The probability for a given leaf to see its connection dropped is $\varepsilon = \frac{sE}{s} = E$. We have

$$\begin{aligned}
\varepsilon &= \sum_{j=\texttt{max\_load}}^{m} (j - \texttt{max\_load}) \cdot \binom{m}{j}\left(\frac{1}{s}\right)^j\left(1-\frac{1}{s}\right)^{m-j} \\
&\le \sum_{j=\texttt{max\_load}}^{m} (j - \texttt{max\_load}) \cdot \left(\frac{m\cdot e}{j}\right)^j\left(\frac{1}{s}\right)^j \cdot 1 && \text{since } \binom{m}{j} \le \left(\frac{m\cdot e}{j}\right)^j \\
&\le \sum_{j=\texttt{max\_load}}^{m} (j - \texttt{max\_load}) \cdot \left(\frac{\alpha\cdot e}{j}\right)^j && \frac{m}{s} \le \alpha \text{ by definition of } \alpha \\
&\le \sum_{j=\texttt{max\_load}}^{\infty} (j - \texttt{max\_load}) \cdot \left(\frac{\texttt{max\_load}}{j} \cdot e^{-(2+\eta)\frac{\ln \texttt{max\_load}}{\texttt{max\_load}}}\right)^j && \text{bound on } \alpha \\
&\le \sum_{\kappa=0}^{\infty} \kappa \cdot 1 \cdot e^{-(\kappa+\texttt{max\_load})(2+\eta)\frac{\ln \texttt{max\_load}}{\texttt{max\_load}}} && \texttt{max\_load} \le j \\
&\le e^{-(2+\eta)\ln \texttt{max\_load}} \int_0^\infty x e^{-x(2+\eta)\frac{\ln \texttt{max\_load}}{\texttt{max\_load}}} dx \\
&= \left(\frac{1}{\texttt{max\_load}}\right)^{2+\eta} \cdot \left(\frac{\texttt{max\_load}}{(2+\eta)\ln \texttt{max\_load}}\right)^2 && \int_0^\infty x e^{-\beta x} dx = \beta^{-2} \\
&\le \frac{1}{4\ln^2 \texttt{max\_load}} \cdot \frac{1}{(\texttt{max\_load})^\eta}
\end{aligned}$$

□

Proof of Theorem 2. For the purpose of an upper-bound on the algorithm failure probability, we may ignore the successes of first phase acquisitions and only consider the second phases. Recall, p is the probability that a leaf is dead. Let $p' = p + 2\varepsilon$. We justify below that p' is an upper-bound on the probability that a request to a leaf is not answered during the second phase, either because the leaf is dead, or because the leaf is overloaded.

Some leaves may receive `max_load` or more in-coming messages during the first acquisition phase. These leaves may be unwilling to serve up to `max_load` requests during the second acquisition phase because they may exceed their load quota of `max_load` connections for each phase. We do not assume that leaves are able to distinguish between first and second phase acquisition messages, therefore they may serve too much during the first phase. Since these leaves may fail to answer requests during the second acquisition, we mark them as dead. Although the number of these leaves may be computed, for convenience we shall use $\varepsilon|\mathcal{D}_u|$ as a rough upper-bound, yielding an increase of ε in the failure probability of targets. Applying Lemma 1 yields another increase, by ε as well, of the probability of targets failing to answer a query.

The probability that a leaf set to acquire the value from a given sub-tree T succeeds is at least $(1-p)\cdot(1-p')$. The probability that a super-leaf fails to acquire the value of T is at most $(1-(1-p)(1-p'))^{k_2} \leq (p+p')^{k_2}$ when there are $k_2 = \frac{(1+\rho)}{2}\ln n$ leaves set to acquire the value from T.

Taking a union bound on all branches whose value a super-leaf needs to acquire, for all super-leaves and for all rounds, we get an upper-bound on the failure probability of the algorithm: $p_{\text{fail}} \leq \left(2b\frac{n}{l}h(p+p')^{k_2}\right)$. Note that $bh/l \leq 1/(1+\rho) \leq 1$. We have

$$p_{\text{fail}} \leq 2e^{\ln n - \frac{(1+\rho)\ln n|\ln(2(p+\varepsilon))|}{2}} = \frac{2}{n^{\frac{|\ln(2(p+\varepsilon))|}{2}-1}} \cdot \frac{1}{\left(n^{|\ln(2(p+\varepsilon))|}\right)^{\frac{\rho}{2}}} \qquad \square$$

Proof of Theorem 3. Consider a virtual node u. There are $S(u)$ super-leaves trying to acquire u's value. Each of them fails to do so during the first phase with a probability upper-bounded by $1/\ln n$ (see proof of Theorem 4). Denote by M the number of super-leaves that need to run a second acquisition phase and by μ the expected value of M. Bound M using a Chernoff bound:

$$\Pr[M \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$

Note that $\mu \leq S(u)/\ln n$. Set $(1+\delta)\mu = e^2 S(u)/\ln n$, which guarantees that $\frac{e}{1+\delta} \leq \frac{1}{e}$, and we have

$$P_1 = \Pr[M \geq \frac{e^2 S(u)}{\ln n}] \leq e^{-\frac{e^2 S(u)}{\ln n}}$$

Define P_1 to be the probability of exceeding this bound. We want P_1 small enough that the probability of some vnode exceeding the bound is smaller than P_{fail}, the algorithm failure probability. There are at most $2n/lb$ vnodes satisfying requirement (2). It is sufficient to have $P_1 2n/lb \leq P_{\text{fail}}$, which can be rewritten as

$$e^{-\frac{e^2 S(u)}{\ln n}} * 2n \leq 2ne^{-\frac{(1+\rho)\ln n|\ln(2(p+\varepsilon))|}{2}}$$

equivalent to Equation 2. This shows that with sufficiently high probability, all super-leaves learn the subtree value during the first acquisition phase, that is, $\text{acq}(u) = k_1 S(u)$ $\square$

Proof of Theorem 4. The number N_1 of messages sent during all first acquisition phases is $\mathcal{O}(\frac{n}{l}2bhk_1)$ where $k_1 = \ln\ln n/|\ln(2p+\varepsilon)|$ is the number of leaves that are dedicated to any given subtree. Simplifying, we get

$$N_1 = \mathcal{O}(n)$$

A second phase acquisition is run only if the first phase one is unsuccessful. A first phase acquisition fails with probability upper-bounded by $(2p+\varepsilon)^{k_1} = 1/\ln n$. We use a Chernoff bound to bound the number M of these unsuccessful first phase acquisitions:

$$\Pr[M \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$

We have $\mu \leq 2bhn/(l \cdot \ln n) \leq 2n/((1+\rho)\ln n)$. Set $(1+\delta)\mu = 2e^2n/((1+\rho)\ln n)$, which guarantees that $\frac{e}{1+\delta} \leq \frac{1}{e}$, and we have

$$\Pr[M \geq \frac{2e^2n}{(1+\rho)\ln n}] \leq e^{-\frac{n}{(1+\rho)\ln n}} \leq e^{-n/2}$$

The last inequality holds for sufficiently large n.

With high probability, the number N_2 of messages sent during all second phase acquisitions is upper-bounded by $\frac{2e^2n}{(1+\rho)\ln n} \cdot \frac{2l}{b}$, that is

$$N_2 = \mathcal{O}(n)$$

Finally, the total number N of messages sent to non-(super-leaf)-local leaves is $N = N_1 + N_2$:

$$N = \mathcal{O}(n)$$ □

Proof of Theorem 5. Assuming no structural changes, the query requests are of size m and the answers of size $\mathcal{A}gg + k_2 + m = \mathcal{O}(\mathcal{A}gg + m)$. In the case of a split, the answer size is $\mathcal{A}gg + 2k_2 + m = \mathcal{O}(\mathcal{A}gg + m)$.

In the case of merging or rebalancing between vnode u and v, k_2 sized information needs to be communicated for each child of u that v needs to know, and of each child of v that u needs to know. There are at most $4b$ of these. The total information exchanged is of size $\mathcal{O}(\mathcal{A}gg + m + bk_2) = \mathcal{O}(\mathcal{A}gg + m + l)$. □