Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

AVCOL: Availability-aware information aggregation in large distributed systems under uncollaborative behavior ☆,☆☆

Ramsés Morales *, Indranil Gupta *

Department of Computer Science, University of Illinois at Urbana-Champaign, Siebel Center, 201 N Goodwin Ave., Urbana, IL 61801, United States

A R T I C L E   I N F O

Article history:
Available online 25 March 2009

Keywords:
Aggregation
Availability-aware
Probabilistic protocol
Predicates
Selfish
Colluding
Overlay

A B S T R A C T

Aggregation of system-wide information in large-scale distributed systems, such as p2p systems and Grids, can be unfairly influenced by nodes that are selfish, colluding with each other, or are offline most of the time. We present AVCOL, which uses probabilistic and gossip-style techniques to provide availability-aware aggregation. Concretely, AVCOL is the first aggregation system that: (1) implements any (arbitrary) global predicate that explicitly specifies any node's probability of inclusion in the global aggregate, as a mathematical function of that node's availability (i.e., percentage time online); (2) probabilistically tolerates large numbers of selfish nodes and large groups of colluders; and (3) scales well with hundreds to thousands of nodes. AVCOL uses several unique design decisions: per-aggregation tree construction where nodes are allowed a limited but flexible probabilistic choice of parents or children, probabilistic aggregation along trees, and auditing of nodes both during aggregation as well as in gossip-style (i.e., periodically). We have implemented AVCOL, and we experimentally evaluated it using real-life churn traces. Our evaluation and our mathematical analysis show that AVCOL satisfies arbitrary predicates, scales well, and withstands a variety of selfish and colluding attacks.

1. Introduction

Distributed applications aggregate various kinds of data from large populations of nodes. Resource utilization information is collected about nodes in order to enable resource discovery for Grid applications [22,31]. Statistics of system performance are collected [20,33], e.g., max, min, top-k, or bottom-k of CPU utilization. Votes may be collected from nodes, and the majority of answers used to make a go-no-go decision, e.g., for leader election or replication [21].

The above systems use aggregation within the network in order to scalably and efficiently compute the aggregate, and deliver it to a sink node. However, in environments where nodes have varying degrees of contribution to the system, one often desires to collect *biased aggregates* so that nodes that have contributed more to the system have a bigger say in the final aggregate. In such a biased aggregation, for each node, the probability that a global aggregate will include that node's own value (henceforth we call this *inclusion probability*), increases with that node's contribution to the system. Notice that the inclusion probability for a node is calculated only across global aggregates initiated while that node is online. Such biased aggregation can be useful to mitigate freeloading [1] by ensuring that nodes that contribute comparatively less to the system, influence the aggregate less.

In this paper, we consider a specific type of contribution, namely node *availability*, defined as the *fraction of time that the node is online*. We allow the inclusion probability of a node's value to be specified as a mathematical function of that node's availability. This relation is a *global predicate* that the application deployer would desire applied to all nodes in the system. We denote this global

☆ This work was supported in part by NSF CAREER Grant CNS-0448246 and in part by NSF ITR Grant CMS-0427089.
☆☆ This work is previously unpublished.
* Corresponding authors. Tel.: +1 217 244 8433 (R. Morales).
E-mail addresses: rvmorale@cs.uiuc.edu (R. Morales), indy@cs.uiuc.edu (I. Gupta).

predicate as f, and we focus only on monotonically non-decreasing predicates. For instance, the deployer might specify a linear predicate, i.e., the inclusion probability of each node x is $f(x) = av(x)$, where $av(x)$ is that node's availability. As another example, a quadratic predicate may be desired, e.g., $f(x) = (av(x))^2$, or a bimodal predicate, e.g., `if` $(av(x) > 0.5)\ f(x) = 1.0$ `else` $f(x) = 0.0$.

Unlike other approaches that implicitly scale node benefit approximately according to its contribution [13,30], our approach allows us to *explicitly* specify this relation as a mathematical function, thus giving a better control over the quality of aggregation. This control enables the use of the aggregation protocol for various purposes. For instance, one can calculate the average available disk space throughout the system, by using the linear predicate along with the "average" aggregation function over the disk space attribute at nodes. If concerns over data durability increase, then the previous aggregation could use the quadratic predicate, thus resulting in a disk space measurement more biased towards what's available at higher availability nodes. Another example is using the bimodal (or quadratic) predicate along with the "min" aggregation function over ids of nodes. This produces a leader election protocol where only high availability nodes can become leaders. In general the bimodal or quadratic predicate can be used to penalize low availability nodes – compared to the use of the linear predicate – and thus provide incentive for them to improve their availability.

The problem of using local and distributed actions at nodes to achieve an *arbitrary* and *emergent* global predicate is a challenging one. There is a need to scale to systems with hundreds or thousands of nodes, as well as to withstand churn, i.e., arrival, departure and failure of nodes. Further challenges come from the fact that nodes may be uncollaborative. This means that: (1) many nodes may be *selfish – a selfish node takes actions that increases its own inclusion probability, independent of its availability*, (2) groups of nodes may be *colluding – they aim to increase their own inclusion probabilities, independent of their availabilities*. Our uncollaborative model is restricted to increasing one's inclusion probability only, and not Byzantine behaviors such as arbitrarily modifying the value of an aggregate or influencing other non-colluding node's inclusion probability.

Selfish and colluding behavior can arise from node gaming [27] or multiple administrative domains (MADs) [2]. This behavior can adversely influence the predicate satisfaction in any solution to our aggregation problem. For instance, in the above examples, selfish nodes may end up unfairly biasing the measured average available disk space. A group of colluders may end up influencing the leader election to always elect one of them as leader (regardless of their availability).

We present AVCOL, an availability-aware aggregation service that implements arbitrary global predicates for biased aggregation. AVCOL works in environments where nodes may be selfish or colluding. AVCOL uses a novel combination of four techniques: (1) building aggregation trees on-demand, where nodes select parents (or children) based on availability, (2) restricting the choice of valid parents (or children) based on a consistent condition, (3) probabilistic forwarding of child data up to parents at each internal tree node, and (4) periodic (i.e., gossip-style) and per-aggregation auditing to verify correct node behavior and prevent collusion. AVCOL can be seen as incorporating availability dependence with a probabilistic aggregation approach.

AVCOL leverages an availability monitoring service, e.g., [24], a distributed partial membership protocol, e.g., [11,32,16,24], and knowledge about the probability distribution of node availability in the system, e.g., [4]. We analyze the latency and reliability of AVCOL's aggregation trees, as well as predicate satisfaction at each node. We also present experimental results from a simulation driven by traces of availability variation from real deployed systems, e.g., the Overnet p2p system.

In our previous work, we have built decentralized protocols that implement global predicates for multicast [26] and membership [23], by leveraging our availability monitoring service [24]. The availability-aware aggregation problem addressed in this current paper is a natural follow-up, and extends the idea of global predicates to the aggregation problem. This problem requires an entirely new set of design techniques.

We present our assumptions and problem statement in Section 1.1 and related work in Section 1.2. The probabilistic aggregation in AVCOL trees is described in Section 2, while Section 3 discusses how trees are constructed in spite of selfish and colluding nodes. Section 4 presents our auditing scheme, and we present experimental results in Section 5. We conclude in Section 6.

1.1. Assumptions and problem statement

We make the following assumptions:

(1) Aggregations occur in rounds, called *epochs*. Each epoch is uniquely identified by using the sink node's id and a signed epoch number. Epochs could be initiated either (a) asynchronously, initiated by the sink, or (b) periodically, at synchronized times across all nodes (e.g., helped by NTP). We support both these options. Epochs do not overlap, and inter-epoch time intervals are larger than the typical time to finish an aggregation.

(2) Each aggregation epoch is associated with a sink node which desires to calculate the aggregate. We assume henceforth for simplicity that the same sink node is used in all epochs; our algorithms work even when the sink differs across epochs. We will also assume that the sink is (i) always online, i.e., has an availability of 1.0, and (ii) is not selfish or collusive with any other node. These assumptions are reasonable because we want aggregation anytime, and at a trusted sink node.

(3) The aggregation statistic desired by the sink is partially aggregatable within the network, i.e., the tree is used for *in-network aggregation*. In other words, akin to [22,31,33], we assume that combining two partial aggregates into another aggregate, does not increase the size of the message. Some partially aggregatable functions are *top-k, bottom-k, max, min, count, sum*, and *average* (aggregated as sum and count).

(4) Each node has a unique id, and can send messages to any other node. In order to bound latency of aggregation, we assume that a message to a correct (alive) node is received within a time bound. We assume each node can sign messages, and signatures can be verified – without this assumption nodes may masquerade as multiple other nodes and launch a Sybil Attack [9].
(5) The number of online nodes, N (a system parameter), is stable and changes within a small constant factor in a timeframe of weeks. This assumption is justified, even under system churn. For instance, in the Overnet system [4] the online node population size varies by a factor of 2 over a week and by a factor of 3 over a month. Furthermore, [6,28] show that the Gnutella system size varies within a factor of 2 per day and per month, and [14] shows that in p2p streaming systems the size varies within a factor of 9 per day and per week. Thus, we can set N to an approximate value in this range, and it can be updated infrequently (e.g., once a month) without hurting scalability. The estimate size can be determined distributedly by existing protocols such as [19].
(6) The node availability PDF remains fairly stable across time. Just like N, this has been shown to be stable in several deployed p2p systems [28]. Thus, it can be measured and used as a system-wide parameter that would be updated infrequently (e.g., once a month), without affecting scalability. This measurement can be done by the availability monitoring service.

1.1.1. Global predicate, and informal problem statement

The desired global predicate, that relates a node x's availability $av(x)$ to the inclusion probability for its data in an aggregate, is denoted by the function $f : [0,1] \rightarrow [0,1]$. We make two assumptions about f: (1) f is monotonically non-decreasing, i.e., if $av(x) > av(y)$ for two nodes x, y, then it is true that $f(av(x)) \geqslant f(av(y))$; (2) $f(1.0) = 1.0$, i.e., a node that is always online will desire to have its data appear in all aggregates. For instance, this is true at the sink node.

The problem we address is then, informally, as follows: given an arbitrary desired global predicate f, design an aggregation protocol so that for each node x, x's contributed value(s) appears in a fraction $f(av(x))$ of the global aggregates at the sink, calculated only across epochs during which x is online.

We would like to achieve this in a *uncollaborative* setting where nodes may be selfish and colluding, and also join, leave, rejoin, and silently fail from the system. A selfish or colluding node attempts to maximize the inclusion probability of its own value in a global aggregate, but without affecting other nodes' inclusion probabilities, i.e., selfish or colluding nodes *are not malicious or Byzantine*. In other words, a node deviates from the specified protocol behavior only when the deviation improves the inclusion probability of either itself, or some of its colluders. Thus, selfish nodes may execute local actions, while colluding nodes may use friends, all to increase their own inclusion probabilities, e.g., by double forwarding of own values. However, nodes never maliciously modify their own values or partial aggregates. We assume an arbitrary number of selfish nodes in the system. We also assume that nodes collude in *groups*, where all pairs of nodes in the same group collude, with the size of the groups being large.

1.1.2. Leveraged services

In order to solve this problem, we leverage two services in a black-box manner: (1) a distributed availability monitoring service [24], and (2) a decentralized probabilistically-shuffled membership protocol [11,16,24]. The distributed availability monitoring service keeps track of the availability of nodes, and allows any node's availability to be queried. The reported availability could be either raw, aged, or window-based (recent). We assume a *consistent* availability monitoring service, i.e., simultaneous queries (e.g., from multiple nodes) for availability of a given node all return the same value. Our implementation uses the AVMON decentralized monitoring system [24], and our experiments measure the effect of any inconsistencies arising from this use. AVMON's overhead is fully distributed and our experiments show it is small [24]. We will elaborate on the decentralized shuffled membership protocol in Section 4.2, where it is first used by our design.

These two leveraged services are themselves resilient to uncollaborative nodes. AVMON reports accurate availabilities in spite of large numbers of uncollaborative nodes. As Section 4.2 shows, we use the membership protocol only for selecting children and parents in the aggregation tree – thus an uncollaborative node cannot increase its own inclusion probability by tampering with the membership.

1.2. Related work

Centralized aggregation schemes based on user scripts or CoMon-like tools [34] collect a lot of information periodically (e.g., once every 10 min) from all nodes, maintaining these in a queriable database. Decentralized aggregation schemes scale better by using in-network aggregation. Many of these build aggregation trees either based on domain layout (e.g., Astrolabe [31] or Ganglia [22]), or by using a structured overlay (e.g., SDIMS [33], PIER [15], or [3]), or randomly on demand (e.g., MON [20]), or based on other techniques. Robust aggregation can be done either via gossip [17,18] or via multiple paths in sensor networks [25].

However, none of these systems above have addressed the effect of selfish or colluding nodes. Similar to many decentralized approaches, AVCOL builds per-aggregation trees. Yet, unlike them, AVCOL innovates in being the first to satisfy explicit availability-based predicates.

Game theoretic techniques have been applied for systems with arbitrary rational nodes [27], yet they are often too complex and bandwidth-consuming for large distributed systems. The BAR model [2] considers Byzantine, altruistic and rational nodes, but has not been applied to the aggregation problem. In addition, BAR allows rational nodes only to be selfish, but not colluding. While AVCOL does not consider Byzantine (malicious) attacks, it does address aggregation under selfish and colluding behavior.

While traditional protocols typically provide a deterministic bound on the number of attackers, e.g., [5], AVCOL tolerates an arbitrary number of selfish nodes. In addition, it provides a *probabilistic* tolerance to large numbers of colluders in the system. Finally, auditing mechanisms have been used to ensure replica correctness in spite of attacks and bit-rot in LOCKSS [21], as well as for detection of Byzantine behavior in PeerReview [12]. Similar to PeerReview, AVCOL reports selfish and colluding nodes via signed non-repudiable proofs.

2. Probabilistic aggregation in AVCOL trees

We first describe how AVCOL trees aggregate data in order to satisfy a given global predicate *f*. While Section 3 will describe how these trees are constructed in order to combat selfish and colluding nodes, the tree aggregation itself is agnostic to such uncollaborative nodes. In other words, the current section assumes no nodes are selfish or colluding.

AVCOL trees are built so that each node *x*'s tree parent has an availability $\geqslant av(x)$. In other words, if a node *y* is a tree parent of a node *x*, then it is true that $av(y) \geqslant av(x)$. Notice that any node in the system can find a prospective parent with a higher or equal availability than itself, since in the worst case it can go to the sink node which has an availability of 1.0. Inductively, this implies that an AVCOL tree can be built to cover all nodes in the system. Section 3 describes tree construction; we now focus on the aggregation and predicate satisfaction. Fig. 1 illustrates an example aggregation tree, and we elaborate below.

Each AVCOL node *x* uses the following probabilistic aggregation while passing data up towards the sink. If node *x* is a leaf in the tree, it sends a message to its parent containing its own value. If node *x* is an internal node, it waits to hear from all of its children. Each child *c* reports an aggregate (denoted as $AG(c)$) for the subtree rooted at *c*. Node *x* then forwards to its own parent a partial aggregate that: (1) includes *x*'s own value with probability 1.0, and

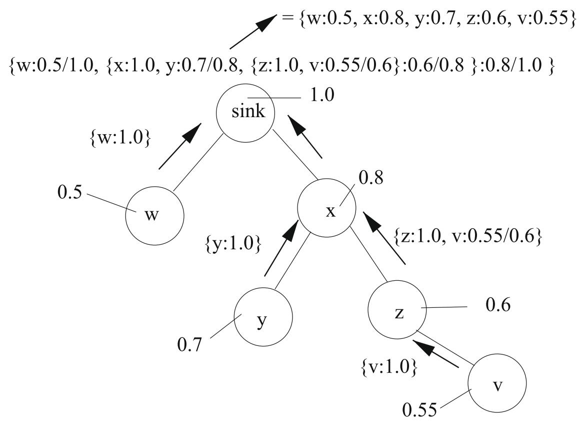
= {w:0.5, x:0.8, y:0.7, z:0.6, v:0.55}
{w:0.5/1.0, {x:1.0, y:0.7/0.8, {z:1.0, v:0.55/0.6}:0.6/0.8 }:0.8/1.0 }
sink
1.0
{w:1.0}
0.5
w
x
0.8
{y:1.0}
{z:1.0, v:0.55/0.6}
y
0.7
z
0.6
{v:1.0}
v
0.55

Fig. 1. Example aggregation tree. $f(av(node))$ is shown for each node, and messages should be read as {datavalue:probability}, with probabilities multiplicative. Final message at top shows resultant inclusion probabilities for each node in the global aggregate. Notice that joining the sink directly as child does not increase inclusion probability.

(2) for each child *c*, includes $AG(c)$ with probability $\frac{f(av(c))}{f(av(x))}$. Notice that this latter quantity is $\leqslant 1$ as a parent's availability is never lower than a child's. In doing this aggregation, the node can use in-network aggregation to calculate a compact partial aggregation (e.g., sum, count, for avg.). The sink node executes step (2) as well, and uses the resulting aggregate as the final answer.

Theorem 2.1. *Consider an epoch during which no nodes join, leave or fail from the system. For any node x that is online and in the tree during this epoch, its own value appears in the global aggregate reported at the sink node S, with probability* $f(av(x))$.

Proof. The proof is by induction on the distance of node *x* from the sink. For the base case, notice that the sink node *S*'s data is included with probability $1.0 = f(av(S))$. Now for any other node *x* with parent *P*, *P* will include *x*'s value in the aggregate sent to *P*'s parent, with probability $\frac{f(av(x))}{f(av(P))}$. If this does happen, then *x*'s data will always accompany *P*'s own value (which is passed up to *P*'s parent with probability 1.0), either all the way up to the sink or until it is probabilistically dropped at some ancestor of *P*. Thus, by induction, since *P*'s own value will appear with probability $f(av(P))$ in the global aggregate, the probability that *x*'s own value will appear in the global aggregate at sink *S* is $= f(av(P)) \cdot \frac{f(av(x))}{f(av(P))} = f(av(x))$. Note that this result holds even if the node is a direct child of the sink node. □

3. AVCOL tree construction

In a realistic setting with node churn, and with selfish and colluding nodes, the static AVCOL trees of Section 2 may be ineffective. This is due to many reasons. Firstly, if parent–child relationships are static, then for each node *x* that is offline during an epoch, all of *x*'s tree descendants will have their values not included in the global aggregate. This will happen for all epochs when *x* is offline. Secondly, during any epoch, a node *x* may send its data to more than one parent, thus potentially increasing *x*'s inclusion probability via multiple counting. These additional parents may be either colluders of *x* (who will pass on to their parents, *x*'s data), or just unaware that *x* is selfishly sending duplicates.

AVCOL addresses these problems by dynamically constructing per-epoch aggregation trees. It works by: (I) providing each node a flexible choice of parents (or children) for each aggregation epoch; (II) limiting this choice to a small set of *valid* parents (or children) that are selected based on a consistent condition; and (III) running audit operations at the valid parents of each node to verify its correct behavior. We describe these operations in the following sections. Intuitively though, design choice (I) builds trees on the fly for each epoch – nodes try to "route around" offline parents, and manage to create a path to the sink with high probability (w.h.p.). (II) reduces the probability that two colluding nodes will be allowed to be parent and child. Finally, (III) ensures that colluding nodes, and nodes sending data to multiple parents during an epoch, are eventually caught and blackmarked.

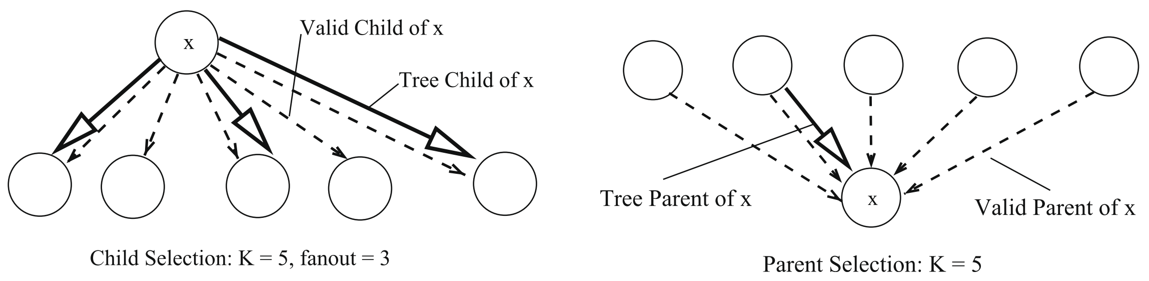

Fig. 2. Child selection (left) and parent selection (right) example at node *x*.

AVCOL has two alternative flavors – *child selection* where nodes select children (Section 3.1), and *parent selection* where nodes select select parents (Section 3.2). Child selection and parent selection are illustrated with an example in Fig. 2. We describe these selection criteria next, and present their analysis. Subsequently, Section 4 discusses how nodes discover children or parents, and the auditing mechanism.

3.1. Child selection approach

In this variant, each node *x* selects and discovers multiple valid children in the tree. The validity of a prospective child node *y* depends only on the ids and availabilities of *x* and *y*, and is independent of everything else in the system. Then, during any given epoch, node *x* selects as its *tree children*, *fanout* random nodes among its valid children. It collects aggregates from these tree children using the probabilistic forwarding of Section 2. *x* is free to choose different tree children in subsequent epochs. We first describe the validity condition for child selection, and then the per-epoch aggregation.

3.1.1. Selection criteria for valid children

At a node *x*, a remote node *y* is a *valid child* if and only if the following consistent condition is true:

$$(\text{i})av(x) \geqslant av(y), \text{and (ii) also}$$
$$H(x,y) < min\left\{\frac{K}{N}\cdot\frac{1}{cdfav(av(x))}, 1.0\right\}$$

(details to follow). Conversely, *x* is said to be a *valid parent* of *y* if both these conditions are true. Here, *H* is a consistent hash function with range in the interval $[0,1]$. For instance, *H* could be either SHA-1 or MD-5, with outputs normalized to $[0,1]$. *x* and *y* used within the hash function are the bitstrings representing the ids of these nodes, e.g., their ip address + port.

As described by our system model in Section 1.1, *N* is a fixed parameter, and is set to the estimated number of online nodes in the system. *K* is a small fixed parameter, set to the expected number of valid children, typically $K = \theta(\log(N))$. *cdfav*() is the fraction of online nodes with an availability $\leqslant av(x)$, i.e., it is the cumulative distribution function of node availability. The value of *cdfav*() can be obtained from the probability distribution function (PDF) of the node availability across the system. Recall that the PDF can be measured by the availability monitoring service, and that it changes infrequently (see Section 1.1).

The discovery of the above valid children is discussed in Section 4.2; we now analyze the properties of the selection scheme itself.

Theorem 3.1.1. *At a given node x, the above child selection scheme has the following desirable properties: (a) choice of valid children is uniformly at random from among all nodes with availability lower than* $av(x)$, *(b) consistency of the parent–child relation in spite of any system changes extraneous to x and y, and (c) verifiability of this relation at any third node. Further, the expected number of valid children for any node is* $O(K)$. *Finally, if* $K = \theta(\log(N))$, *then the graph created by the valid parent–child relationships is strongly connected w.h.p.*

Proof. The child selection is random since it picks a node *y* (among those with $av(y) \leqslant av(x)$) with a uniform probability $min\left\{\frac{K}{N}\cdot\frac{1}{cdfav(av(x))}, 1.0\right\}$. The consistent hash function ensures that the valid parent–child relationship depends only on the ids and availability of x, y. The validity is verifiable at any third node, which can check the consistent condition using only the ids of x, y, and their availabilities fetched from the availability monitoring service. □

Next, the expected number of nodes with availability lower than $av(x)$ is $cdfav(av(x))\cdot N$, and thus the expected number of valid children of *x* is:

$$\leqslant \left(\frac{K}{N}\frac{1}{cdfav(av(x))}\right)(cdfav(av(x))\cdot N) = K.$$

Finally, theoretical results about random digraphs (if avg. degree >1, there is a giant component) [7,10] can be used to show that if $K = \theta(\log(N))$, then the graph created by the valid parent–child relationships is strongly connected w.h.p.

3.1.2. Per-epoch aggregation

During each epoch, the sink initiates the protocol by sending a signed tree request to itself. Suppose a node *x* sends *y* a signed tree request – *y* first checks if *x* is a valid parent (according to the consistent condition), or if $x = y =$ the sink. If neither is true, *y* reports *x* as an uncollaborative node; the received tree request serves as non-repudiable proof of this accusation. If *x* is a valid parent, *y* responds to it with an acknowledgement; further, *y* will refuse any future tree requests during this epoch, sending negative acknowledgements to such requests (unless *y* is selfish or colluding). Once a signed tree request has been received,

y randomly chooses up to *fanout* among its valid children, after verifying that they are still online and still satisfy the consistency condition. y then sends these potential children signed tree requests from itself.

Since recruiting children does not affect a node's own inclusion probability, a node has no incentive *not* to recruit valid children. However, a colluding node may have an incentive to either always (i.e., in every epoch) select as tree child a colluder node that already happens to be a valid child, or to select as tree child a colluder node that is not a valid child. This enables the colluder (i.e., the tree child) to send its data to either a non-valid parent, or to multiple parents. These two behaviors are addressed by the auditing mechanisms described in Section 4.

In the aggregation for a given epoch, a node is a leaf if it either sends no tree requests (it might not know any valid children), or it received negative acknowledgements to all its requests, or it timed out before receiving an aggregate from any child. A leaf merely sends its own value, after signing it, to its parent. Otherwise, consider an internal node that has received aggregates from one or more valid children that it had sent tree requests to. This internal node waits until either all its tree children have replied or a local timeout has elapsed. This node then *audits* these children (described in Section 4.1); if this succeeds, x then uses the probabilistic forwarding rule previously described in Section 2 to send the aggregated data to its parent. This data is signed before being sent. All timeout values used at the leaf and internal nodes are $(c \cdot \log(N))$; this is motivated by our latency calculations below. This timeout is local and based on when the aggregation started at this node.

Once again, a selfish/colluding node has no incentive to time out early, or to maliciously drop any of its children's reported aggregates, since this does not affect the inclusion probability of itself or any of its colluders. However, a node that colludes with some of its children (e.g., a non-valid child, or a child sending data to multiple parents) may want to prevent this collusion from being discovered by either not auditing, or by lying about the auditing result. Section 4 catches these behaviors.

A couple of closing notes – first, although a node can have at most one parent per epoch, this parent can be different across epochs. This provides fault-tolerance. Second, for either asynchronous or periodic aggregation, if a correct node did not receive any tree request from a parent during a given epoch, then it never participates in the tree for that epoch. We call such nodes as *orphans*, and their presence reduces the coverage of the tree, which we analyze next.

3.1.3. Analysis

Child selection basically creates a random spanning tree among the nodes, with the sink as the tree root. We now calculate the *coverage* C of the tree, i.e., the probability of a node being reached from the sink by the tree during an epoch. Coverage of a node is important since it is a prerequisite for the satisfaction of the global predicate at that node.

In order to isolate tree performance from uncollaborative behavior, we analyze coverage when all nodes are correct (i.e., not selfish or colluding). Suppose node x has K valid parents, and let $NC = 1 - C$ = non-coverage probability, so that $NC \ll 1$. Node x is not covered during an epoch only if either (i) all of its valid parents are offline, or (ii) none of its valid parents are covered, or (iii) none of the valid parents choose x as a child for this epoch. If a is the expected availability across x's valid parents (a thus increases as $av(x)$ rises), then we can write these three probabilities as: $NC \leqslant (1-a)^K + NC^K + (1-NC^K)\cdot(1-\frac{fanout}{K})^K$. Now, let us assume $K = m_1 \cdot \log(N)$, $fanout = m_2 \cdot \log(N)$, and $m_2 \leqslant m_1$, where m_1, m_2 are constants. The results of gossip-based tree building in [10] indicate that under these conditions, $NC \ll 1$. Eliminating small terms:

$$NC \leqslant (1-a)^K + NC^K + (1-NC^K)\cdot e^{-fanout}$$
$$\simeq (1-a)^K + e^{-fanout} = \frac{1}{N^{m_1 \cdot \log\frac{1}{1-a}}} + \frac{1}{N^{m_2}}.$$

Hence, if one uses high enough constant values for m_1, m_2, the coverage of the child selection approach is $C = \Omega(1 - o(1))$. Specifically, choosing $m_1 \cdot log\frac{1}{1-a}, m_2 \geqslant 1$ yields coverage probability $C = \Omega(1 - \frac{1}{N})$.

A caveat in the above analysis is that as $av(x)$ increases, so does a, and thus the coverage probability of node x may decrease. Thus, if one were to set the parameter value of $K = \log_{\frac{1}{1-a^*}}(N)$ to be same at all nodes, one should plug in a high enough value of a^* in order to still obtain good coverage for such nodes. Higher availability nodes that see themselves excluded from too many epochs (i.e., not satisfying the global predicate) can use *sink redirection*, i.e., such nodes can request the sink and become its direct tree children. Due to probabilistic forwarding at the sink (Section 2), this redirection still satisfies the predicate at these nodes.

Finally, one can use results from gossip multicast [8] to show that the *latency of aggregation* – i.e., time from aggregate initiation to completion – is $O(\log(N))$. The choice of timeout in waiting for children's answers is motivated by this calculation.

3.2. Parent selection approach

In comparison to child selection, parent selection has each node (1) select and discover multiple valid parents based on a consistent condition, and (2) for each epoch, select one of its valid parents as a tree parent, in order to forward data from itself and its children.

3.2.1. Selection criteria for valid parents

At a node x, a remote node y is a valid parent if and only if the following consistent condition is true:

$$\text{(i)}\, av(x) \leqslant av(y), \text{and (ii) also}$$
$$H(x,y) < min\left\{\frac{K}{N}\cdot\frac{1}{1-cdfav(av(x))}, 1.0\right\}.$$

Conversely, x is said to be a valid child of y if both above conditions are true. The definition of H and $cdfav()$ remain the same as in Section 3.1. Similar to child selection, we can prove:

Theorem 3.2.1. *At a given node x, the above parent selection scheme has the following desirable properties: (a) choice of valid children is uniformly at random from among all nodes with availability higher than av(x), (b) consistency of* *the parent–child relation in spite of any system changes extraneous to x and y, and (c) verifiability of this relation at any third node. Further, the expected number of valid parents for any node is $O(K)$. Finally, if $K = \theta(\log(N))$, then the graph created by the valid parent–child relationships is strongly connected w.h.p.*

3.2.2. Per-epoch aggregation

The aggregation tree is built starting from the sink. Each node selects a *tree parent* from among its valid parents. In order to do this, each node maintains a local, binary state variable for each epoch. The node can be in one of two states – *intree* or *notintree*. At the start of an epoch, only the sink is *intree*, while all other nodes are *notintree*. We now describe the tree construction separately for each of the periodic and asynchronous problem settings.

First, for periodic aggregation, at the start of the synchronized epoch, each node sends one tree request to each of its valid parents. A node selects as its tree parent the first reply received to any of these requests, and then changes to *intree*. The node in turn also queues all its received tree requests from valid children, and acknowledges them all when its own state first turns to *intree*. On the other hand, in the asynchronous aggregation setting, each node periodically asks each valid parent for the latest epoch numbers for which the parent is *intree*. For a given epoch, whenever node x first discovers any of its valid parents that happens to be *intree*, then x does the following: (i) selects this as its tree parent, (ii) sends it a tree request, and (iii) marks itself (i.e., x) as *intree*. In a nutshell then, both the periodic and asynchronous schemes have the effect of the *intree* state propagating top-down from the sink node.

Notice that selfish nodes have no incentive to lie about their *intree*/*notintree* state, since recruiting children does not increase one's inclusion probability. Further, all messages carrying this state are signed so that they can be used to prove valid parent–child relationships. Thus, the only downside of this protocol is that it potentially makes it possible for selfish nodes to acquire non-valid parents – Section 4 catches these.

If a node does not find any of its valid parents marked as *intree* before a timeout expires, then it uses *sink redirection*, i.e., it sends a tree request directly to the sink. The timeout is chosen as $c \cdot \log(N)$, is local to the node, and starts at the same time as the current aggregation epoch. The sink may receive many tree requests; however, having the sink as tree parent does not improve a node's inclusion probability (due to the analysis in Theorem 2.1).

Finally, a node that has succeeded in recruiting a tree parent performs aggregation by using the probabilistic forwarding of Section 2. A leaf node waits awhile (timeout chosen as $c \cdot \log(N)$ since it recruited the parent) before sending its value up to its parent. An internal tree node waits until either the timeout elapses, or until all its tree children – those that requested a parent and were acknowledged – have responded. It then audits the tree children (described in Section 4); if this succeeds, it uses the probabilistic forwarding from Section 2. All aggregate messages sent to a tree parent are signed.

Similar to child selection, a node x has no incentive to time out early or drop any received aggregate from a tree child. However, such a node x may still lie about a colluder tree child that is a non-valid child, or about a valid child sending data to x and multiple tree parents during the same epoch. These are addressed by Section 4.1.

3.2.3. Analysis

First, one can show that the expected height of the parent selection tree during an epoch is $O(\log(N))$ [8]. Second, these trees obtain good coverage even without sink redirection, as we show below.

Theorem 3.2.2. *Consider an epoch during which no nodes join or leave. Suppose that each online node has at least one path to the sink via valid child to parent pointers. Then each online node is included in that epoch's tree, i.e., coverage is 100%.*

Proof. The proof follows by induction on the distance from the sink, i.e., minimum length of a path from the node to the sink. The base case is true since any valid child of a sink node will become its tree child soon after the sink turns *intree*. Consider a node x at distance i from the sink – at least one of its valid parents is at distance $(i-1)$, will turn *intree* during this epoch (induction hypothesis), and thus x will also turn *intree*. □

Thus, parent selection provides 100% coverage, which is better than the probabilistic coverage of child selection. However, this comes at the cost of higher bandwidth overhead, as potential parents need to be polled each epoch for an acknowledgement (to find out whether any has been marked *intree*). Keep in mind that in parent selection nodes only keep track of potential parents. We will evaluate and compare the child and parent selection strategies in Section 5.

4. Auditing and discovery

This section describes how nodes carry out per-epoch audit operations (Section 4.1), how nodes discover children or parents (Section 4.2), and periodic audit operations (Section 4.3). The audit operations detect selfish nodes and small collusion groups eventually, while probabilistically preventing large collusion groups from having an impact.

4.1. Per-aggregation auditing

We describe per-epoch audit operations that aim to detect pairs of colluding nodes which are not valid parent–children (i.e., do not satisfy the consistent condition), yet acted as tree parent–children during the given epoch. These per-aggregation audit operations apply to both the child and parent selection flavors of AVCOL.

Consider a node x forwarding a partial aggregate message with its contribution and contributions from its tree children – the ones included by the predicate – to its tree parent P. This message is signed by node x. Denote the aggregates from x's tree children $C_1, C_2, \ldots, C_m$, respectively as $AG_1, AG_2, \ldots, AG_m$, x's own value as $v(x)$, and the aggregate calculated from the above $(m+1)$ values as AG_x. Denote the corresponding signed messages received from each C_i as $\{AG_i\}_{C_i}$. Then, the message sent by x is: $\{AG_x, C_1, \{AG_1\}_{C_1}, \ldots, C_m, \{AG_m\}_{C_m}, v(x)\}_x$.

When parent P receives this signed message, it first verifies whether x is indeed a valid child of P. This auditing is done by fetching the availability of x from the availability service, and checking the consistency condition. If the check fails, P can report x as an incorrect node, with the above signed message included in the report as a verifiable, non-repudiable and non-forgeable proof.[1] If x is a valid child, P next verifies for each C_i, whether x is a valid parent of C_i. If all C_i's pass this test, x then checks whether AG_x is indeed a correct aggregate derived from $AG_1, \ldots, AG_m, v(x)$. If this succeeds as well, then P probabilistically forwards AG_x to its own tree parent, as described earlier in Section 2. However, if P finds some C_i to not satisfy the hash consistency condition, then it reports both x and C_i as colluding nodes. The signed aggregation message received from x is a verifiable, non-repudiable and non-forgeable proof of collusion. Formally:

Lemma 4.1.1. *If node x is not a valid parent of y, but acts as its tree parent during some epoch, and if none of the colluders of x are its valid parents, then x and y will be reported.*

Let us call the above as a *2-hop auditing scheme* since each node checks for validity of tree parent–child relationships up to 2 hops below it in the tree. In general, the above scheme can be generalized to *t-hop auditing* by passing along signed aggregation messages from all its tree descendants up to t hops below it, and subjecting it to consistency checks. We can thus generalize:

Theorem 4.1.2. *If no group of colluding nodes has more than t nodes, then the t-hop auditing scheme will detect at least two such colluding nodes.*

In addition to the above deterministic detection for small collusion groups, AVCOL also provides *probabilistic* tolerance to collusions. We show this via two theorems:

Theorem 4.1.3. *Given a group of M colluding nodes selected randomly from across the node population, and* $M \ll \sqrt[3]{\frac{N}{K}}$ (i.e., $\frac{K \cdot M^3}{N} = o(1)$), *it is true w.h.p. that no pair of colluders is related as a valid parent and child.*

Proof. We present the proof for only parent selection; a similar proof holds for the child selection case. First, let the system have N nodes. Rank all the nodes from 0 to $(N-1)$; a lower rank is assigned to a node with a higher availability. Now, consider a colluder node x in the group that is at rank r. Due to parent selection, and since there are an expected $\left(\frac{r}{N} \cdot M\right)$ colluders with higher availability than x, the expected number of colluders that are valid parents of x is: $min\{\frac{K}{N} \cdot \frac{N}{r}, 1.0\}\left(\frac{r}{N} \cdot M\right) = min\{\frac{K \cdot M}{N}, \frac{r \cdot M}{N}\}$. □

Now, the probability of a colluder node having rank $r < K$ is $= \frac{K}{N}$, thus the probability that no colluders in the group have rank $<K$ is $= \left(1 - \frac{K}{N}\right)^M \geqslant 1 - \frac{M \cdot K}{N} = 1 - o(1)$. Here we have used the fact that $(1-x)^a \geqslant 1 - a \cdot x$ for $x \in [0,1), a > 0$. Thus, there are no colluders with rank $<K$ w.h.p. This means we can ignore the second term $\left(\frac{r \cdot M}{N}\right)$ within the *min* expression above.

Next, for a colluder node x at rank r, and another colluder node y at rank i ($>r$), the probability that x is a valid parent of y is $= \frac{\frac{K \cdot M}{N}}{i}$. Since there are an expected $\left(\frac{N-r}{N} \cdot M\right)$ colluders with higher rank than node x, the probability that *no colluder nodes* will be valid children of x is:

$$\Pi_{colluders, i>r}\left(1 - \frac{K \cdot M}{N \cdot i}\right) \geqslant \left(1 - \frac{K \cdot M}{N \cdot r}\right)^{\frac{N-r}{N} \cdot M} \geqslant \left(1 - \frac{K \cdot M^2}{N^2} \cdot \frac{N-r}{r}\right).$$

Finally, the probability no node pair from among the M colluders are related as valid parent–children is:

$$\Pi_{colluders, r}\left(1 - \frac{K \cdot M^2}{N^2} \cdot \frac{N-r}{N}\right) \geqslant \left(1 - \frac{K \cdot M^2}{N}\right)^M \geqslant \left(1 - \frac{K \cdot M^3}{N}\right) = 1 - o(1).$$

Theorem 4.1.4. *Suppose there is a group of M colluding nodes selected randomly across the node population, with* $M \ll \sqrt{3}\frac{N}{K}$ (i.e., $\frac{K \cdot M^3}{N} = o(1)$). *Then, during an epoch where at least one pair among these colludes, the 2-hop auditing scheme discovers at least one colluding pair w.h.p.*

Proof. From Theorem 4.1.3, it is true w.h.p. that no node pair is a valid parent–child. Thus, if some nodes collude as parent–child during an epoch, there will be a *chain* of such colluders created by tree child to tree parent relationships. Yet, since the sink is a non-colluder, we are assured that any such chain will end in a non-colluder node. This non-colluder will, due to the described auditing, detect collusions of the two chain nodes right below it in the tree. □

4.2. Discovering valid children/parents

In order to discover valid children in child selection (respectively valid parents in parent selection), we leverage a decentralized shuffling membership service such as CYCLON [32], T-Man [16], or AVMON's coarse view [24]. A decentralized shuffling protocol service maintains, at each node, a partial and weakly consistent list of nodes, in such a way that the list is (i) a random selection and (ii) is constantly changed (shuffled). This maintenance depends on gossiping, i.e., a node will periodically contact a peer chosen uniformly at random from the network, and then exchange information. The entries in the list are updated lazily, thus some may be stale and point to offline nodes. The probabilistic-shuffling ensures that given two nodes x and y that are online for long enough, the entry for node y will eventually appear in the membership list at node x.

We discuss below the actions for the child selection variant; parent selection is analogous. Each node maintains a

[1] E.g., Such reports can either be sent to a central auditor, or fed into a reputation system.

list of valid children at all times. Each node frequently and periodically executes the following two actions. (1) It iterates through its current membership list and evaluates the consistent condition on each entry (see Sections 3.1, 3.2). Any entries that satisfy the consistent condition are added to the list of valid children. (2) The node also re-evaluates the consistent condition on its current list of valid children; any entries no longer satisfying the condition are removed from the valid children list. Notice that these steps involve querying the availability service for each of the checked nodes.

Since a similar mechanism was presented and analyzed in [23], we do not reproduce its analysis here. It suffices to mention that a membership list size of $\sqrt{N}$ per node achieves quick discovery with reasonably low bandwidth of (105 Bps for $N = 1$ million) and memory (6.3 KB). The average discovery time is 5 h (for $N = 1$ million), which is reasonable given the uptime of nodes in p2p systems (e.g., 20–30% of the nodes observed at any time in a p2p system have an uptime longer than one day [29]), Grids, and PlanetLab. Finally, due to the probabilistic shuffling of the membership list, this mechanism guarantees *eventual discovery* of valid children that satisfy the consistent condition for long enough.

We reiterate that an uncollaborative node cannot manipulate the membership protocol to increase its own inclusion probability. This is because adding or deleting membership entries, or refusing to participate in the membership protocol, only affects the discovery time of valid parents/children, but does not change the consistent selection criteria.

4.3. Periodic auditing

While the per-aggregation audit of Section 4.1 detected colluding nodes not satisfying the consistent condition, it did not detect selfish nodes that during some epoch, send their data to multiple *valid* parents that satisfy the consistent condition. A node may do this in order to increase its own inclusion probability, and the multiple valid parents could either be its colluders or non-colluders. Such behavior is detected by periodic auditing operations, which we describe below. Periodic auditing is initiated by each node asynchronous, lazily and infrequently, and at a much lower frequency than the parent/child discovery of Section 4.2.

In order to enable periodic audits, each node keeps two types of additional state. First, the node keeps a log history of all aggregate messages received from each of its tree children, for all past epochs since the last periodic audit operation it initiated. Second, the node x maintains a list of the *step-parents* of each of its own valid children. In other words, x maintains, for each of its own tree children c that have sent x past aggregates, *all the valid potential tree parents* of node c. For child selection, discovery of step-parents can be started whenever the valid child is discovered. For parent selection, this discovery is started when the first aggregate is received from this child. Notice though that if a child never sends an aggregate to x, it does not really need to be audited.

Step-parent discovery is implemented using gossiping – the step-parents of such valid children c are discovered in a similar manner to Section 4.2, i.e., by frequently and periodically snooping on the membership list at x, and checking (via the consistent condition) whether any of the membership list entries are valid parents of c. In addition, node x keeps this list up to date by periodically checking the consistent condition, purging out entries that are no longer valid step-parents. Due to the probabilistically-shuffled nature of the membership list, this mechanism ensures eventual detection of all step-parents of valid children.

Node x then uses the above additional state to initiate infrequent and periodic audit operations for its valid children. Specifically, for each valid child c, the periodic audit operation is executed as follows. Node x contacts all the known step-parents of c, and sends to each of them a *log-slice for c*. The log-slice for c is a list of aggregates signed by c, and received at x since the latest periodic audit initiated at x. Audit messages are signed by the initiator node x, and contain a sequence number differentiating it from previous audits by x. Each step-parent z upon receiving such an audit message containing a log slice, first verifies whether x is a valid parent of c – if not, z has a verifiable, non-repudiable and non-forgeable proof that x generated a spurious audit. If the verification succeeds, z sends back to x its log-slice for c, i.e., all the aggregation messages that z received from node c since its last audit.

Now, whenever node x hears back from a step-parent of c, it compares the received log-slice with its own log-slice. It checks to see if there is any common epoch across both slices (recall that log entries are signed and thus verifiable). If no epoch is common, node x does nothing. However, if there is a common epoch across the two log-slices, then node x has a verifiable, non-repudiable and non-forgeable proof that node c sent aggregation messages to multiple parents during an epoch.

Finally, whenever node x finds that a node c has stopped being a valid child (e.g., if its availability changed so the consistent condition is no longer true), it immediately audits c, treating it as a periodic audit. This ensures that x is able to audit its remaining log history for c, and subsequently purges this log history.

The overhead of auditing is on expectation $O(K)$ messages. Since Theorem 4.1.3. showed that all the valid parents of a node are not its colluders w.h.p., we have:

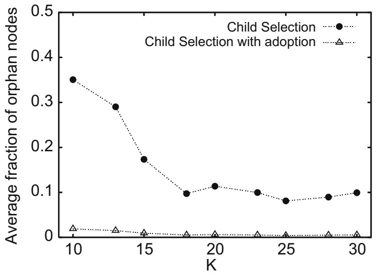

Fig. 3. Fraction of orphan nodes vs. K for child selection, without and with adoption.

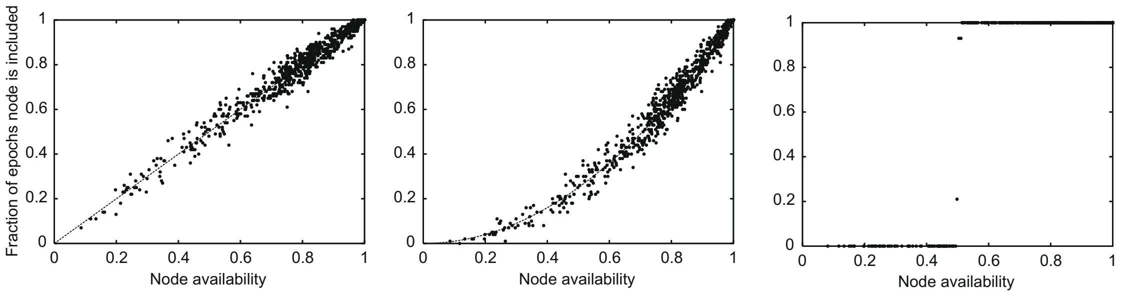

Fig. 4. Probability of inclusion of a node as a function of its availability: linear predicate ($f(x) = av(x)$), quadratic predicate ($f(x) = (av(x))^2$), and bimodal predicate, (`if` $(av(x) > 0.5)$ $f(x) = 1.0$ `else` $f(x) = 0.0$). Line shows predicate, and datapoints are per-node.

Theorem 4.3.1. *Suppose all the valid parents of a node x have discovered each other as step-parents. If x sends data to multiple valid parents during any subsequent epoch, the periodic auditing will eventually detect this behavior.*

In practice, it is possible that a node's availability varies so much that its valid parents are unable to discover each other before they become invalid. Our experiments next evaluate how much real availability traces affect the success of periodic auditing.!

5. Experimental results

We implemented AVCOL in C, and evaluate it using trace-driven discrete-event simulations. AVCOL is built atop AVMON [24], which provides both the availability monitoring service – itself resistant to selfish and colluding nodes – and the probabilistically-shuffled membership protocol that we require (Sections 3 and 4). In order to measure AVCOL's performance under real availability traces, we use the churn traces collected by Bhagwan et al. [4] from the Overnet p2p file sharing network.

5.1. Parameter settings

Unless otherwise noted, all experiments use the following settings. The Overnet churn traces are injected without modification. These traces were collected across a population of 2400 nodes at 20 min intervals, during 7 days, and are injected as such into our simulated system. The availability for 50% of the hosts is 0.3 measured over the 7 days. We use a value of $N = 525$ based on an estimate of the number of online nodes in the trace. Each node maintains a partial and weakly consistent list of other nodes in the system, $\mathscr{L}$, with the number of entries fixed at $|\mathscr{L}| = \lceil\sqrt{N}\rceil = 23$. The underlying system AVMON updates $\mathscr{L}$ periodically every 60 s (Section 4.2) – a longer period could be used, but as [24] shows, AVMON's background bandwidth is only 6.81 Bps for 2000 nodes. AVCOL also maintains, at each node, a list ($\mathscr{V}$) of valid children (if child selection is used), or valid parents (if parent selection is used). List $\mathscr{V}$ is refreshed once every four minutes (Section 4.2). Parameter K is chosen as $\lceil 1.75 \cdot \lceil log_2(N)\rceil\rceil = 18$ (justified in our first experiment). Finally, we set $fanout = K$ in all experiments.

5.2. Node coverage

We evaluate the coverage of our AVCOL trees, considering only child selection and no selfish or colluding nodes. A node is covered if it is reachable from the sink via a path of tree parent–child pointers. Uncovered nodes are said to be orphaned. Coverage improves as we increase the value of K in our algorithm (Section 3); we choose $K = \lceil c \cdot \lceil log_2(N)\rceil\rceil$, and vary c. The upper curve in Fig. 3 shows the percentage of online nodes that are left orphaned as K is varied from 10 to 30. Each datapoint on the plot is an average taken over 200 epochs. It can be seen that beyond $K = 18$ ($c = 1.75$) a plateau is reached and only about 10% nodes are orphans. This motivated our default value for K.

5.2.1. Optimization

The number of orphans can be reduced by sink redirection (Section 3.1). We implement this as *adoption*: if a node has been (1) not covered for at least $\mathscr{O}$ epochs, and (2) no online node is a valid parent[2], then the sink will adopt the node as a child. The lower curve in Fig. 3 shows that with $\mathscr{O} = 10$, fewer than 2% nodes are orphaned. We also observed that in all simulation runs, all orphans were covered within 10 epochs. Finally, we observed that the sink had to adopt only a few orphaned nodes to reach full coverage; this is because an adopted node's descendants are also automatically covered by the adopted node, thus naturally avoiding scalability issues at the sink.

5.3. Satisfying aggregation predicates

The main goal of AVCOL was to satisfy global predicates f relating each node x's inclusion probability to its availability $av(x)$, as $f(av(x))$. Fig. 4 shows the predicate satisfaction for three specific predicates, under parent selection and no selfish or colluding nodes: (1) linear, $f(x) = av(x)$; (2) quadratic, $f(x) = av(x)^2$; (3) bimodal, `if` $(av(x) > 0.5)$ $f(x) = 1.0$ `else` $f(x) = 0.0$. For these plots, the AVCOL system was allowed to warm up for 13 h (simply to allow collection of a long trace for validation), and 200 epochs were generated simultaneously and independently. Each datapoint on these plots is an average over 200 epochs, and cor-

[2] Although this second condition is not required for our algorithm, existence of valid parents can be checked by having parents send periodic heartbeat messages to valid children.

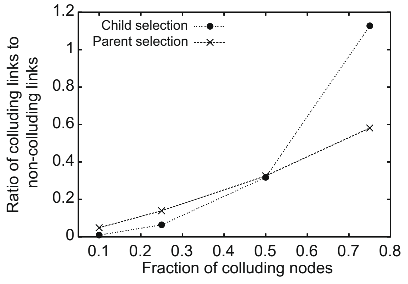

Fig. 5. Ratio of colluding links to non-colluding links as a function of colluder group size.

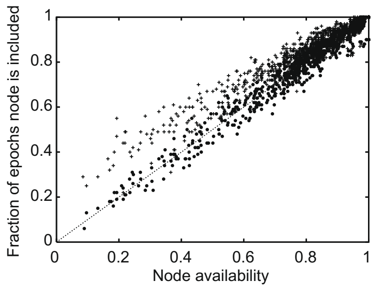

Fig. 6. Linear global predicate with child selection when 10% (dots) and 50% (crosses) of the nodes are colluding.

responds to one node. For each datapoint, the *y*-axis value shows the fraction of the node's online epochs during which its own value was included in the global aggregate at the sink. The *x*-axis value plots the availability of that node as reported by AVMON. The plot shows that in spite of the distributed nature of the aggregation, predicates are satisfied by child selection trees in AVCOL. The results for parent selection were similar, and are not plotted.

5.4. Effect of colluding nodes

We create a colluding group of nodes by randomly selecting a sizable fraction of nodes. Colluders follow valid parent–child relationships and the rest of the AVCOL protocol, except that during each epoch every colluder node: (a) prefers as tree parent (resp. tree children) all colluders that occur among its valid parents (resp. valid children); and (b) includes its colluder tree children's values with probability 1.0 in the aggregate that it passes up to its own tree parent.

First, Fig. 5 plots the effect of colluder group sizes ranging from 10% to 75% of the node population. Even with 25% of the nodes in a colluding group, only about 5% of the valid parent–child links are between colluder pairs (in parent selection). This shows the advantage of using the consistent condition of Section 3. Next, Fig. 6 shows the effect on the linear global predicate of two colluder groups – 10% and 50% of the node population. We make two observations: (1) with 10% of the nodes in a colluding group, the predicate satisfaction is indistinguishable from Fig. 4, and (2) even with 50% nodes in the colluding group, the predicate satisfaction does not degrade much.

5.5. Selfish nodes using multiple parents

We study the effect of "greedy nodes", i.e., selfish nodes that forward data to multiple valid parents. Due to the

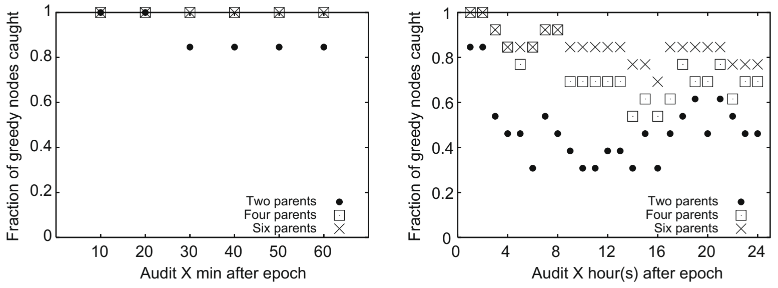

Fig. 7. Fraction of greedy nodes caught by periodic audit, as a function of audit frequency.

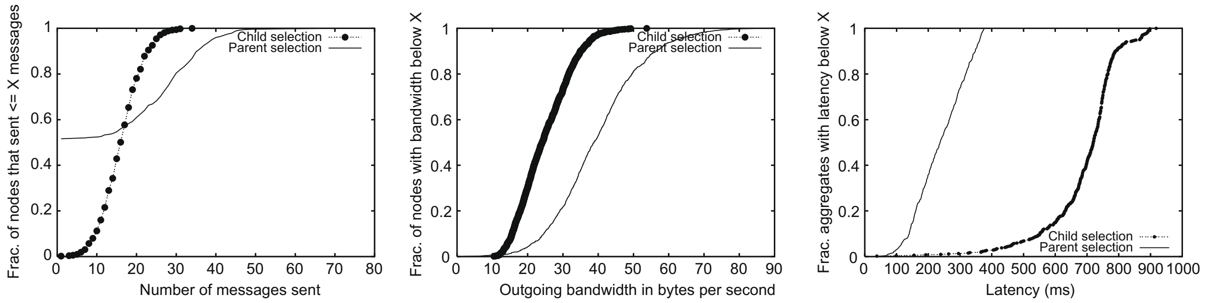

Fig. 8. Left: number of messages per-aggregation. Center: bandwidth due to availability queries to AVMON. Right: latency of parent selection (periodic) vs. child selection (asynchronous).

varying availability of nodes and potentially inconsistent availabilities reported by AVMON, a greedy node may take a while to be detected by the periodic audit operations of Section 4.3 – recall that step-parent discovery, to perform auditing, depends on a gossip-based partial membership list. In this experiment, we select a few greedy nodes and have them send data to multiple valid parents (2, 4, or 6), during just one epoch. We allow AVCOL to function normally, but vary the frequency of periodic auditing. Fig. 7 shows, for child selection, the fraction of greedy nodes caught as a function of the time between the epoch and the next periodic audit. We make these observations: (1) the more greedy a node is (#parents data sent to), the quicker it will be caught, (2) if a greedy node sends data to 4 or more parents, an auditing frequency over once per hour will catch the greedy node, (3) greedy nodes sending to 2 parents or fewer, require a higher auditing frequency (once every 20 min), and (4) lower frequency audits still catch over half the greedy nodes, so a node that chooses to be greedy over several epochs will eventually be caught.

5.6. Bandwidth and latency

We compare child selection against parent selection. Fig. 8 (left) shows the cumulative distribution (CDF) for per-aggregation number of messages. Fig. 8 (center) shows the CDF for background bandwidth, arising from availability queries sent to the leveraged AVMON service. Fig. 8 (right) shows the latencies. Latency for a node is the time taken by its value to reach the sink, measured from the epoch start time. The network latency between each node pair is selected uniformly at random in the interval $[20\,ms, 80\,ms]$ (larger latency than a LAN). We find that: (1) The mean per-aggregation bandwidth in both variants is comparable (15 messages/epoch), however parent selection has a lower median because many nodes are tree leaves and send only 1 message. (2) The median background bandwidth for the two variants is 20–30 Bps. (3) Child selection has higher latency (median: 700 ms) due to tree construction, while parent selection (median: 250 ms) is faster due to periodic aggregation, avoiding the tree construction phase.[3] Finally, AVMON's background bandwidth was calculated as 6.81 Bps for 2000 nodes [24]. These low bandwidth numbers make the system scalable.

6. Conclusions

We have presented AVCOL, the first probabilistic aggregation system to support availability-based global predicates that relate each node's inclusion probability in an aggregate, explicitly to the node's availability. AVCOL's decentralized mechanisms allow arbitrary predicates, and address both selfish nodes and colluding groups of nodes, attempting to increase their inclusion probability. Our analysis of AVCOL shows that it tolerates large numbers of selfish nodes, and large groups of colluding nodes. Our experimental evaluations used real availability variation traces. They showed that aggregation is fast and consumes low bandwidth, several useful predicates can be satisfied in spite of many colluding nodes, and gossip-based auditing catches selfish nodes quickly.

[3] Note that these latencies are significantly lower than the remaining uptime – which measures in minutes and hours – of any one node, at any time, on widely deployed p2p systems [29]. Furthermore, node failure would affect, on expectation, aggregates originating on low availability nodes.

References

[1] E. Adar, B.A. Huberman, Free riding on Gnutella, First Monday 5 (10) (2000).
[2] A.S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, C. Porth, BAR fault tolerance for cooperative services, in: Proceedings of the ACM SOSP, 2005, pp. 45–58.
[3] M. Bawa, H. Garcia-Molina, A. Gionis, R. Motwani, Estimating Aggregates on a Peer-to-Peer Network, Technical Report, Stanford University, 2003.
[4] R. Bhagwan, S. Savage, G. Voelker, Understanding availability, in: Proceedings of the IPTPS, February 2003, pp. 135–140.
[5] M. Castro, B. Liskov, Practical Byzantine fault tolerance and proactive recovery, ACM Transactions on Computer Systems 20 (4) (2002) 398–461.
[6] J. Chu, K. Labonte, B. Levine, Availability and locality measurements of peer-to-peer -+lesystems, in: Proceedings of the SPIE, vol. 4868, 2002.
[7] C. Cooper, A. Frieze, The size of the largest strongly connected component of a random digraph with a given degree sequence, Combinatorics, Probability and Computing 13 (3) (2004) 319–337.
[8] A.J. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, Epidemic algorithms for replicated database maintenance, in Proceedings of the Sixth ACM PODC, 1987, pp. 1–12.
[9] J.R. Douceur, The sybil attack, in: IPTPS'01 Revised Papers from the First International Workshop on Peer-to-Peer Systems, Springer-Verlag, London, UK, 2002. pp. 251–260.
[10] P.T. Eugster, R. Guerraoui, S. Handurukande, A.-M. Kermarrec, P. Kouznetsov, Lightweight probabilistic broadcast. in: Proceedings of the DSN, 2001, pp. 443–452.
[11] A.J. Ganesh, A.-M. Kermarrec, L. Massoulie, SCAMP: peer-to-peer lightweight membership service for large-scale group communication, in: Proceedings of the Third NGC, 2001, pp. 44–55.
[12] A. Haeberlen, P. Kouznetsov, P. Druschel, Peerreview: practical accountability for distributed systems, in: Proceedings of the Third ACM SOSP, 2007, pp. 175–188.
[13] M. Haridasan, I. Jansch-Porto, R. van Renesse, Enforcing fairness in a live-streaming system, in: Proceedings of the ACM MMCN, 2008.
[14] X. Hei, C. Liang, J. Liang, Y. Liu, K.W. Ross, A measurement study of a large-scale p2p iptv system, IEEE Transactions on Multimedia 9 (8) (2007) 1672–1687.
[15] R. Huebsch, J.M. Hellerstein, N. Lanham, B.T. Loo, S. Shenker, I. Stoica, Querying the internet with PIER, in: Proceedings of the VLDB, Springer, Berlin/Heidelberg, 2003, pp. 321–332.
[16] M. Jelasity, O. Babaoglu, T-Man: gossip-based overlay topology management, in: Self-Organising Systems: ESOA, LNCS, vol. 3910, July 2005, pp. 1–15.
[17] M. Jelasity, A. Montresor, Epidemic-style proactive aggregation in large overlay networks, in: Proceedings of the 24th ICDCS, 2004, pp. 102–109.
[18] D. Kempe, A. Dobra, J. Gehrke, Computing aggregate information using gossip, in: Proceedings of the 44th IEEE FOCS, 2003.
[19] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, A. Demers, Active and passive techniques for group size estimation in large-scale and dynamic distributed systems, Elsevier Journal of Systems and Software 80 (10) (2007) 1639–1658.
[20] J. Liang, I. Gupta, K. Nahrstedt, Reliable on-demand management operations for large-scale distributed applications, ACM SIGOPS OSR 41 (5) (2007) 82–88.
[21] P. Maniatis, M. Roussopoulos, T.J. Giuli, D.S.H. Rosenthal, M. Baker, Y. Muliadi, Preserving peer replicas by rate-limited sampled voting, in: Proceedings of the ACM SOSP, 2003, pp. 44–59.
[22] M.L. Massie, B.N. Chun, D.E. Culler, The ganglia distributed monitoring system: design implementation and experience, Parallel Computing 30 (7) (2004) 817–840.
[23] R. Morales, B. Cho, I. Gupta, AVMEM – availability-aware overlays for management operations in non-cooperative distributed systems,

in: Proceedings of the ACM/IFIP/Usenix Middleware, 2007, pp. 266–286.
[24] R. Morales, I. Gupta, AVMON: optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. in: Proceedings of the ICDCS, 2007, p. 55.
[25] S. Nath, P.B. Gibbons, S. Seshan, Z.R. Anderson, Synopsis diffusion for robust aggregation in sensor networks, in: Proceedings of the ACM SenSys, 2004, pp. 250–262.
[26] T. Pongthawornkamol, I. Gupta, AVCast: new approaches for implementing availability-dependent reliability for multicast receivers, in: Proceedings of the IEEE SRDS, 2006, pp. 345–354.
[27] T. Roughgarden, Selfish Routing and the Price of Anarchy, first ed., MIT Press, 2005.
[28] D. Stutzbach, R. Rejaie, Characterizing unstructured overlay topologies in modern p2p file-sharing systems, in: Proceedings of the IMC, 2005, pp. 49–62.
[29] D. Stutzbach, R. Rejaie, Understanding churn in peer-to-peer networks, in: Proceedings of the Sixth ACM SIGCOMM IMC, 2006, pp. 189–202.
[30] Y.-W. Sung, M. Bishop, S.G. Rao, Enabling contribution awareness in an overlay broadcasting system, in: Proceedings of the ACM SIGCOMM, 2006, pp. 411–422.
[31] R. van Renesse, K. Birman, W. Vogels, Astrolabe: a robust and scalable technology for distributed system monitoring, management, and data mining, ACM Transactions on Computer Systems 21 (2) (2003) 164–206.
[32] S. Voulgaris, D. Gavidia, M. van Steen, CYCLON: inexpensive membership management for unstructured P2P overlays, Journal of Network and Systems Management 13 (2) (2005) 197–217.
[33] P. Yalagandula, M. Dahlin, A scalable distributed information management system, in: Proceedings of the ACM SIGCOMM, 2004, pp. 379–290.
[34] CoMon, <http://comon.cs.princeton.edu/>.

Ramsés Morales is currently a PhD student in the Computer Science department of the University of Illinois at Urbana-Champaign. He received his MS in Computer Science from the same university in 2005 supported by a Fulbright Fellowship. Research interests include P2P systems, Distributed Protocols with Self-*Behavior, and Grid Computing.

Indranil Gupta completed his PhD in Computer Science from Cornell University in 2004. Indranil received the NSF CAREER award in 2005 and the Xerox Award in 2008. He has previously worked in IBM Research and Microsoft Research. He obtained his B.Tech (Computer Science) from the IIT-Madras, in 1998. He is a member of ACM and IEEE.