Known Unknowns in Large-Scale System Monitoring

Navendu Jain, Dmitry Kit, Prince Mahajan, Praveen Yalagandula[†], Mike Dahlin, and Yin Zhang
Department of Computer Sciences *[†]Hewlett-Packard Labs*
University of Texas at Austin *Palo Alto, CA*

ABSTRACT

This paper addresses a central challenge in PRISM, a large-scale distributed monitoring system: coping with the uncertainties and ambiguities introduced by network and node failures. In particular, in a large scale monitoring system, such failures interact badly with techniques needed for scalability like hierarchy, arithmetic filtering, and temporal batching. For example, if a monitoring subtree is silent over an interval, it is difficult to distinguish between two cases: (a) the subtree has sent no updates because the inputs have not significantly changed or (b) the inputs have significantly changed but the subtree is unable to transmit its report. As a result, reported results can be arbitrarily far from their true values.

To address this challenge PRISM introduces *Network Imprecision* (NI), a new metric to characterize accuracy despite node failures, network disruptions, and system reconfigurations. PRISM leverages NI to flag potentially inaccurate results, allowing applications to differentiate between known-correct and likely-erroneous results as well as to correct distorted results by applying several redundancy techniques. Evaluation of our PRISM prototype shows that NI effectively flags inaccurate query results while incurring low overheads, and we find that using NI to automatically select the best results can reduce the inaccuracy in a PRISM-based monitoring service by nearly a factor of five.

1. INTRODUCTION

This paper describes how PRISM[1] uses *Network Imprecision* (NI) to enable scalable monitoring. The key idea of NI is that because no system can guarantee to always provide the "right" answer [12, 32], it instead must report the extent to which a calculation could have been disrupted by node and network problems. Intuitively, NI represents a "stability flag" indicating whether the underlying network is stable or not.

Scalable system monitoring and distributed stream processing are fundamental abstractions for large-scale networked systems. They serve as basic building blocks for applications such as network monitoring and management [6, 16, 20, 41], financial applications [2], resource scheduling [17, 40], efficient multicast [38], sensor networks [17, 40], resource management [40], and bandwidth provisioning [8] that may scale to thousands or millions of dynamic attributes (e.g., per-flow or per-object state) spanning tens of thousands of nodes.

Three techniques are vital for a monitoring system's scalability: (1) *hierarchical aggregation* [17, 20, 38, 40] allows a node to access detailed views of nearby information and summary views of global information, (2) *arithmetic filtering* [19–21, 25, 34, 45] caches recent reports and only transmits new information if it differs by some numeric threshold (e.g., $\pm$ 10%) from the cached report, thus trading a bounded small approximation error for a significant load reduction, and (3) *temporal batching* [19, 21, 25, 34, 38] combines multiple updates that arrive near one another in time into a single network message. Each of these techniques can reduce monitoring overheads by an order of magnitude or more [19, 20, 25, 40].

As important as these techniques are for reducing load, they interact badly with network and node failures: a monitoring system that uses any of these scalability techniques risks reporting highly inaccurate results.

- First, when a monitoring system uses arithmetic filtering, if a subtree or node is silent over an interval, the system must distinguish two cases: (a) the subtree or node has sent no updates because the inputs have not significantly changed from the cached values or (b) the inputs have significantly changed but the subtree or node is unable to transmit its report.
- Second, under temporal batching there are windows of time in which a short disruption can block a large batch of updates (e.g., when a disruption delays transmission of the combined update), resulting in large inaccuracies due to staleness in a monitoring system's reports.
- Third, in a hierarchical monitoring system, the impact of failures is made worse by the *amplification effect* [24]: if a non-leaf node fails, then the entire subtree rooted at that node can be affected. For example, failure of a level-3 node in a degree-8 aggregation tree can interrupt updates from 512 (8^3) leaf node sensors.

These effects can be significant. For example, in one network monitoring application, we observed that more than half of all reports differed from the ground truth at the inputs by more than 60%.

To address this challenge, PRISM introduces *Network Imprecision* (NI), a new metric for characterizing the inconsistency in query results due to network instability. In particular, given that no system can guarantee to always provide the "right" answer [12, 32], NI attaches an

[1]PRecision-Integrated Scalable Monitoring

assessment of the current network state with each query result. Then, a query result with low NI is highly likely to reflect reality, but an answer with a high value of NI indicates a low system confidence in that query result—the network is unstable, hence the result should not be trusted.

We design, implement, and evaluate PRISM, which makes use of arithmetic filtering, temporal batching, and hierarchy for scalability, and which leverages NI to effectively safeguard accuracy by, for example, (1) inferring an approximate confidence interval for the number of sensor inputs contributing to a query result, (2) differentiating between correct and erroneous results based on their NI, or (3) correcting distorted results by applying redundancy techniques and then using NI to automatically select the best results.

A key challenge is implementing NI efficiently. Because a given failure has different effects on different aggregation trees embedded in our scalable DHT [40], the NI reported with an attribute must be specific to that attribute's tree. Unfortunately, detecting missing updates due to failures, delays, and reconfigurations requires frequent active probing of paths within a tree. As a result, naive probing can limit the scalability of the system, subverting the benefits of arithmetic filtering and temporal batching. Therefore, to provide a topology-aware implementation of NI that scales to tens of thousands of nodes and millions of attributes, PRISM introduces a novel *dual-tree prefix aggregation* construct that exploits symmetry in its DHT-based aggregation topology to reduce the per-node overhead of tracking the n distinct NI values relevant to n aggregation trees in an n-node DHT from $O(n)$ to $O(\log n)$ messages per unit time. For a 1024-node system, dual tree prefix aggregation reduces the per node cost of tracking NI from a prohibitive 100 messages per second to about 5 messages per second.

The most important benefit of NI is the ability to quantify and improve confidence in the accuracy of outputs by addressing network instability and the amplification effect: we observe that for monitoring systems that ignore NI, half of their reports can differ from the truth by more than 60%. Conversely, by using NI metrics to automatically select the best of four redundant aggregation results, we can reduce the observed worst-case inaccuracy by nearly a factor of five.

This paper makes four contributions. First, we present Network Imprecision, a new consistency metric that characterizes the impact of network instability on aggregate query results. Second, we provide a scalable implementation of NI via dual-tree prefix aggregation. Third, our evaluation demonstrates that NI is vital for enabling scalable aggregation: a system that ignores NI can often silently report arbitrarily incorrect results. Finally, we demonstrate how different applications can leverage NI to detect distorted results and take corrective action.

The rest of this paper is organized as follows. Section 2 provides background on the scalable DHT-based aggregation technologies and approximation techniques that underlie PRISM. Section 3 describes the NI abstraction and explains how different applications use NI. Section 4 presents how to scalably compute the NI metrics. Section 5 presents the case-study applications that we build to drive the development and evaluation of PRISM. Section 6 presents the experimental evaluation of our system. Finally, Section 7 discusses related work, and Section 8 concludes.

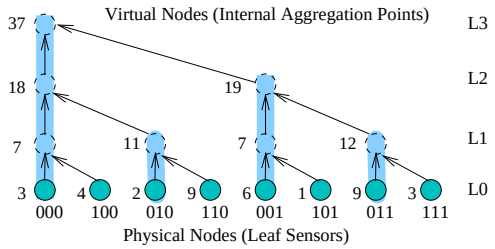


Figure 1: The aggregation tree for key 000 in an eight node system. Also shown are the aggregate values for a simple SUM() aggregation function.

2. BACKGROUND AND PROBLEM

PRISM achieves scalability by combining three well-known techniques: DHT-based hierarchical aggregation, arithmetic filtering, and temporal batching. In this section, we briefly describe these techniques, define the guarantees PRISM must enforce, and illustrate the challenges to meeting these guarantees in a large-scale system. Additional details of PRISM's implementation of these three concepts are available elsewhere [19, 20, 40].

DHT-based hierarchical aggregation. PRISM's aggregation abstraction defines a tree spanning all nodes in the system. As Figure 1 illustrates, each physical node in the system is a leaf and each subtree represents a logical group of nodes[2]. An internal non-leaf node, which we call a *virtual node*, is simulated by one or more physical nodes at the leaves of the subtree rooted at the virtual node.

PRISM leverages DHTs [28–30, 35, 46] to construct a forest of aggregation trees and maps different attributes to different trees [4, 10, 28, 31, 40] for scalability and load balancing. DHT systems assign a long (e.g., 160 bits), random ID to each node and define a routing algorithm to send a request for key k to a node $root_k$ such that the union of paths from all nodes forms a tree *DHTtree*$_k$ rooted at the node *root*$_k$. By aggregating an attribute with key k = hash(attribute) along the aggregation tree corre-

[2]Note that logical groups can correspond to administrative domains (e.g., department or university) or groups of nodes within a domain (e.g., a /28 subnet with 14 hosts on a LAN in the CS department) [15, 40].

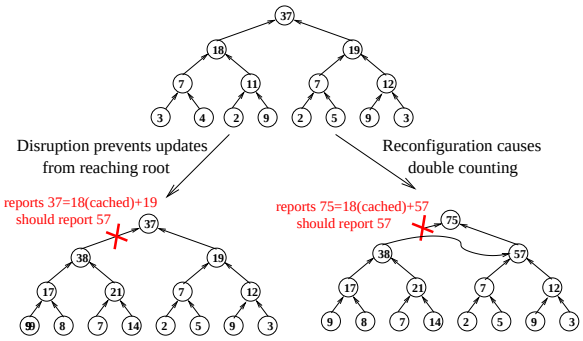


Figure 2: Dynamically-constructed aggregation hierarchies raise two challenges for guaranteeing the accuracy of reported results: the *failure amplification effect* and *double counting* caused by reconfiguration.

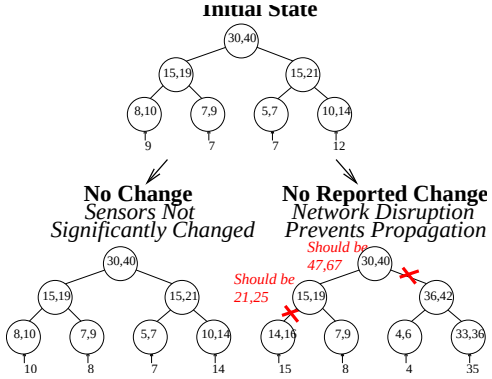


Figure 3: Arithmetic filtering makes it difficult to determine if a subtree's silence is because the subtree has nothing to report or is unreachable.

sponding to $DHTtree_k$, different attributes are load balanced across different trees. Studies suggest that this approach can provide aggregation that scales to large numbers of nodes and attributes [4, 10, 28, 31, 40].

Unfortunately, as Figure 2 illustrates, hierarchical aggregation imperils correctness in two ways. First, a failure of a single node or network path can prevent updates from a large collection of leaves from reaching the root, amplifying the effect of the failure [24]. Second, node and network failures can trigger DHT reconfigurations that move a subtree from one attachment point to another, causing the subtree's inputs to be double-counted by the aggregation function for some period of time.

Arithmetic Imprecision (AI). Arithmetic imprecision bounds the difference between the reported aggregate value and the true value. In PRISM, each aggregation function reports a range $\{V_{min}, V_{max}\}$ in which the true aggregate value, computed by applying that aggregation function across the inputs, lies [19].

Allowing such arithmetic imprecision enables numeric filtering: a subtree need not transmit an update unless the update drives the aggregation value outside the range it last reported to its parent. Numerous systems have found that small amounts of arithmetic imprecision can greatly reduce overheads [19–21, 25, 34, 38, 45], and PRISM allows the size of the range to be set and enforced on a per-attribute basis to enable adaptive precision-overhead tradeoffs [20].

Unfortunately as Figure 3 illustrates, arithmetic filtering raises a challenge for correctness: if a subtree is silent, it is difficult for the system to distinguish between two cases. Either the subtree has sent no updates because the inputs have not significantly changed from the cached values or the inputs have significantly changed but the subtree is unable to transmit its report.

Temporal Imprecision (TI). Temporal imprecision bounds the delay from when an event/update occurs until it is reported. In PRISM, each attribute has a TI bound, and

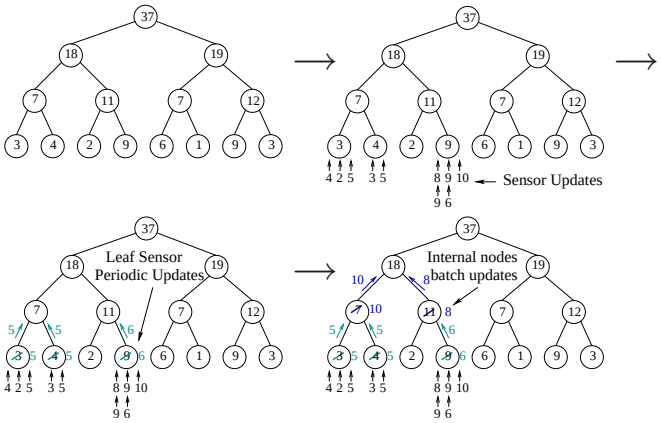


Figure 4: Temporal batching allows leaf sensors to condense a series of updates into a periodic report and allows internal nodes to combine updates from different subtrees before transmitting them further.

to meet this bound, the system must ensure that updates propagate from the leaves to the root in the allotted time.

As Figure 4 illustrates, TI allows PRISM to use temporal batching: a set of updates at a leaf sensor are condensed into a periodic report or a set of updates that arrive at an internal node over a time interval are combined before being sent further up the tree [19].

Of course, an attribute's TI guarantee can only be ensured if there is a *good path* from the leaf to the root. A good path is a path whose processing and network propagation times fall within some pre-specified delay budget [19]. Note that node/network failures/delays can cause a path to no longer be good thereby preventing the system from meeting its TI guarantees. Furthermore, when a system batches a large group of updates together, a short network or node failure can cause a large error. For example, suppose a system is enforcing TI=60s for an attribute, and suppose that an aggregation node near the root has collected 59 seconds worth of updates from

its descendents but then loses its connection to the root node for a few seconds. That short disruption can cause the system to violate its TI guarantees for a large number of updates spread across a large number of nodes.

3. NI ABSTRACTION AND APPLICATION

In this section we describe the NI abstraction that characterizes the consistency of aggregate query results. We first present the NI metrics that measure the stability of the system in Section 3.1 and then use a simple example in Section 3.2 to illustrate how these NI metrics characterize the potential inaccuracy introduced by network disruptions. Section 3.3 then illustrates how this information characterizing the stability of the system can be used by applications to improve their accuracy. All of the discussions in this section assume that NI is provided by an oracle; in Section 4, we describe how to compute the NI metrics accurately and efficiently.

3.1 NI metrics

The NI abstraction is driven by two fundamental properties of any large-scale monitoring system. First, no monitoring system can guarantee to always provide perfect consistency in a dynamic environment [12,32]. Thus, NI can at best flag incorrect results as untrustworthy. Second, large distributed systems may never be 100% stable therefore simply flagging results as "right" or "wrong" does not suffice. Instead, NI should provide a scale to characterize how stable the network is during the computation of a query result. This information gives applications the flexibility to choose the desired query result consistency by setting NI thresholds appropriately based on application requirements.

To quantify system stability, NI provides three metrics: N_{all}, $N_{reachable}$, and N_{dup}.

- N_{all} is an estimate of the total number of nodes in the system.
- $N_{reachable}$ is a lower bound on the number of nodes whose *recent* input values are guaranteed to be reflected in the query result. Recency is defined by the TI guarantees the system provides for the attribute. For example, if the TI is 60 seconds, then $N_{all} - N_{reachable}$ is the number of inputs whose values may be stale by more than 60 seconds.
- N_{dup} provides an upper bound on the number of nodes whose input contribution to a result may be doubly-counted. Double-counting can occur when reconfiguration of an aggregation tree's topology causes a leaf node or a subtree rooted at an internal node to switch to a new parent while its old parent retains the node's or subtree's input as soft state until a timeout.

These three metrics characterize the consistency of a query result: $N_{reachable}$ close to N_{all} and a low N_{dup} indicates that results reflect most inputs and are likely to be useful. For example, N_{all} = 100, $N_{reachable}$ = 99, and N_{dup} = 0 implies that the query result accounts for all but one node and hence is highly likely to be accurate.[3] Conversely, query answers with high values of $N_{all} - N_{reachable}$ or N_{dup} suggest that the network is unstable; hence the results should not be trusted. For example, if $N_{reachable}$ = 30 and N_{dup} = 20, the query result may be missing inputs from 70% nodes and further may double-count inputs of 20% nodes.

3.2 Example

Here, we present how these three metrics provide the NI abstraction using a simple example.

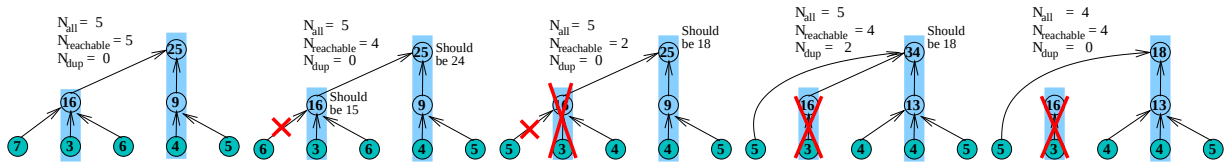
Consider the aggregation tree computing a SUM aggregate across 5 physical nodes in Figure 5(a); Figures 5(b)-(e) illustrate how the NI metrics track the tree's changing topology because of failures and reconfigurations. For simplicity, we compute a SUM aggregate under an AI filtering budget of zero (i.e., update propagation is suppressed if and only if the value of an attribute has not changed), and we assume a TI guarantee of TI_{limit} = 10 seconds (i.e., the system attempts to guarantee a maximum staleness of 10 seconds.) Finally, to avoid spurious garbage collection/reconstruction of per-attribute state, PRISM configures the underlying DHT to reconfigure topology if a path is down for a long (e.g., 10-minute) timeout, and internal nodes retain inputs from their children cached as soft state for slightly longer than that amount of time.

Initially, (a) the system is stable; the root reports the correct aggregate value of 25 with $N_{all} = N_{reachable} =$ 5 and $N_{dup} = 0$ implying that all nodes' recent inputs are reflected in the aggregate result with no double-counting.

Then, (b) the input value changes from 7 to 6 at a leaf node, but before sending that update, the node gets disconnected from its parent. Because of soft state caching, the failed node's old input is still reflected in the SUM aggregate, but recent changes at that sensor are not; the root reports 25 but the correct answer is 24. As (b) shows, NI exposes this inconsistency to the application by changing $N_{reachable}$ to 4 within TI_{limit} = 10 seconds of the disruption, indicating that the reported result is based on stale information from at most one node.

Next, we show how NI exposes the failure amplification effect. In (c), a single node failure disconnects the entire subtree rooted at that node. NI reveals this major disruption by reducing $N_{reachable}$ to 2 since only two leaf nodes retain a good path to the root. The root still reports 25 but the correct answer (i.e., what an oracle

[3]The accuracy is dependent on the aggregate function e.g., for MAX, the one missed input might be the maximum. NI provides a mechanism for reporting disruptions and applications determine an appropriate policy for coping with different levels of disruption based on application requirements. Several such policies are discussed in Section 3.3.



(a) Fault-free (b) Node disconnect (c) Amplification (d) Reconfig 1 (e) Reconfig 2

Figure 5: The evolution of $N_{reachable}$, N_{all}, and N_{dup} as nodes fail and the system reconfigures. The values in the center of each circle represent the value of an example SUM aggregate. The vertical blue bars show the virtual nodes corresponding to a given physical node at the leaf.

would compute using the live sensors' values as inputs) is 18. By using NI, the application learns that failures are affecting the result since only 2 out of 5 nodes are reachable. Thus, this report cannot be trusted; the application can either discard it or take corrective actions such as those discussed in Section 3.3.

We now show how NI exposes the effects of overlay reconfiguration. After a timeout (d), the affected leaf nodes detect the failure and switch to a new parent; NI exposes this change by increasing $N_{reachable}$ to 4. But since the nodes' old values may still be cached, NI increases N_{dup} to 2 implying that two nodes' inputs are doubly-counted in the root's answer of 34.

Finally, we show how NI highlights when the system has restabilized. In (e), the system again reaches a stable state—the soft state expires, N_{dup} falls to zero, N_{all} becomes equal to $N_{reachable}$ of 4, and the root reports the correct aggregate value of 18.

3.3 Using NI

PRISM's formulation of NI explicitly separates the basic mechanism for detecting and quantifying NI from the policy of how to minimize the result inaccuracy caused by failures and reconfigurations. This separation is needed because the impact of omitted updates (when $N_{reachable} < N_{all}$) or duplicated updates (when $N_{dup} > 0$) depends on the topology of the aggregation tree (e.g., a leaf node failure may have less impact than an internal node failure), the nature of the aggregation function (e.g., some aggregation functions are insensitive to duplicates [7]), the variability of the sensor inputs (e.g., when inputs change slowly, using a cached update for longer than desired may have a modest impact), and application requirements (e.g., some applications may prize availability over correctness and live with best effort answers while others may prefer not to act when the accuracy of information is suspect.) Therefore, PRISM reports N_{all}, $N_{reachable}$, and N_{dup} and allows applications to evaluate the significance of disruptions and to take application-appropriate actions to manage this impact.

The simple mechanism of providing these three NI metrics is nonetheless powerful—it supports a broad range of techniques for coping with network and node disruptions. We first describe four standard techniques we have implemented: (1) flag incorrect answers, (2) choose the best of several answers, (3) on-demand reaggregation when the reported answer is unacceptable, and (4) probing to determine the numerical contribution of duplicate or stale inputs. We then briefly sketch other ways applications can make use of NI.

- *Filtering or flagging unacceptably uncertain answers*—PRISM's first standard technique is to manage the trade-off between consistency and availability [12] by sacrificing availability: applications report an exception rather than returning an answer when NI exceeds a threshold. Alternatively, applications can be configured to maximize availability by always returning an answer based on the best available information but flagging that answer's quality as high (e.g., $NI_{tot} < 1\%$, where $NI_{tot} = \text{MAX}[\frac{N_{all}-N_{reachable}}{N_{all}}, \frac{N_{dup}}{N_{all}}]$), medium (e.g., $NI_{tot} < 10\%$), or low (e.g., $NI_{tot} \geq 10\%$).

- *Redundant aggregation*—PRISM can aggregate an attribute using k different keys so that one of the keys is likely to find a route around the disruption. Since each key is aggregated using a different tree, each has a different NI associated with it, and the application chooses the result associated with the key that has the smallest NI.

  In PRISM's DHT-based aggregation topology, most nodes in an aggregation tree are near the leaf level. Therefore, a small increase in k significantly reduces the probability that a given failure affects a node near the root level in all k trees. In particular, if there are f independent failures in an ℓ-level d-ary aggregation tree, the expected number of disconnected nodes is $f * (\ell + 1)$ but the standard deviation is high: $f * d^{\frac{\ell}{2}}$. In comparison, by aggregating an attribute along a (small) constant number of trees, then with high probability, the expected number of disconnected nodes due to all failures occurring at level $\leq i$ ($i << \ell; i = 0$ is leaf level) has a mean $f * (i + 1)$ but a smaller standard deviation

$f * d^{\frac{i}{2}}$. E.g., for ℓ = 2, d = 16, f = 10, $i = 1$, aggregating an attribute along $k = 4$ trees decreases deviation from 160 for a single tree to 40; detailed proofs are in the technical report [19]. Later in Section 6.2, we show that by aggregating an attribute up $k = 4$ paths and using NI to choose best the answer, we can reduce inaccuracy by nearly a factor of five.

- *On-demand reaggregation*—given a signal that current results may be affected by significant disruptions, PRISM allows applications to trigger a full on-demand reaggregation to gather current reports (without AI caching or TI buffering) from all available inputs. In particular, if an application receives an answer with unacceptably high N_{dup} or $N_{all} - N_{reachable}$, it issues a "probe" to force all nodes in the aggregation tree to discard their cached data for the attribute and to recompute the result using the current value at all reachable leaf inputs.

- *Determine V_{dup} or V_{stale}*—when N_{dup} or $N_{all} - N_{reachable}$ is high, an application knows that many inputs may be double counted or stale. An application can gain additional information about how the network disruption affects a specific attribute by computing V_{dup} or V_{stale} for that attribute. V_{dup} is the aggregate function applied to all inputs that indicate that they may also be counted in another subtree; for example in Figure 5(d), V_{dup} is 9 from the two nodes on the left that have taken new parents before they are certain that their old parent's soft state has been reclaimed. Similarly, V_{stale} is the aggregate function applied across cached values from unreachable children; in Figure 5(c) V_{stale} is 16, indicating that 16/25 of the sum value comes from nodes that are currently unreachable.

  Since per-attribute V_{dup} and V_{stale} provide more information than the NI metrics, which merely characterize the state of the topology without reference to the aggregation functions or their values, it is natural to ask: Why not always provide V_{dup} and V_{stale} and dispense with the NI metrics entirely? As we will show in Section 4, the NI metrics can be computed efficiently. Conversely, the attribute-specific V_{dup} and V_{stale} metrics must be computed and actively maintained on a per-attribute basis, making them too expensive for indiscriminant use. Given the range of techniques that can make use of the much cheaper NI metrics, PRISM provides them as a general mechanism but allows applications that require (and are willing to pay for) the more detailed V_{dup} and V_{state} information to do so.

For other monitoring applications, it may be useful to apply other domain-specific or application-specific techniques. Examples include

- *Duplicate-insensitive aggregation*—some systems can be designed with duplicate-insensitive aggregation functions where nodes can transmit multiple copies of aggregate values along different paths to guard against failures without affecting the final result. For example, MAX is inherently duplicate-insensitive [21], and duplicate-insensitive approximations of some other functions exist [7, 22, 24].

- *Increasing reported TI*—short bursts of reduced $N_{reachable}$ mean that an aggregated value may not reflect some recent updates. Rather than report a result with low staleness but a high NI (e.g., $NI_{tot} > 20\%$), the system can report an older result with a low NI (e.g., $NI_{tot} < 1\%$) but explicitly increase the TI staleness bound.

- *Statistical Data Analysis*—systems can combine application-level redundancy and statistical inference to estimate the missing values, as well as estimating the process parameters for the model generating those values. E.g., Bayesian inference [33] has been used in a 1-level tree to estimate missing sensor inputs and model parameters in an environmental sensor network.

These examples are illustrative but not comprehensive. Armed with information about the likely quality of a given answer, applications can take a wide range of approaches to protect themselves from network disruptions.

4. COMPUTING NI METRICS

In this section we describe how PRISM computes the three NI metrics. It is important to note that whereas AI and TI are specified and enforced on a per-attribute basis, NI is maintained by the system for each aggregation tree and shared across all attributes mapped to a tree. This arrangement amortizes the cost of maintaining NI.

Although monitoring connectivity to nodes to compute the NI metrics N_{all}, $N_{reachable}$, and N_{dup} appears straightforward—the metrics are all conceptually aggregates across the state of the system—in practice two challenges arise. First, the system must cope with reconfiguration of dynamically constructed aggregation trees; otherwise the aggregate NI values might include reports of disconnected subtrees as well as double count the contribution of rejoined subtrees. Second, the system must scale to large numbers of nodes despite (a) the need for active probing to measure liveness between each parent-child pair and (b) the need to compute distinct NI values for each of the large number of distinct aggregation trees in the underlying DHT forest; otherwise the system will incur excessive monitoring overhead as we show in Section 4.3.

In the rest of this section, we first provide a simple algorithm for computing N_{all} and $N_{reachable}$ for a single, static tree. Then, in Section 4.2 we explain how PRISM

computes N_{dup} to account for dynamically changing aggregation topologies. Later, in Section 4.3 we describe how to scale the approach to a large number of distinct trees constructed by PRISM's DHT framework.

4.1 Single tree, static topology

This section considers calculating N_{all} and $N_{reachable}$ for a single, static-topology aggregation tree.

N_{all} is simply a count of all nodes in the system, which serves as a baseline for evaluating $N_{reachable}$ and N_{dup}. N_{all} is easily computed using PRISM's aggregation abstraction. Each leaf node inserts 1 to the N_{all} aggregate, which has SUM as its aggregation function. Note that even if a node becomes disconnected from the DHT, its contribution to the N_{all} aggregate remains cached as soft state by its ancestors for a long timeout $T_{declareDead}$. $T_{declareDead}$ is set to be longer than the timeout used to trigger topology changes by the underlying DHT so that a node whose parent becomes unreachable will have time to connect to a new parent before being removed from the N_{all} count.

$N_{reachable}$ for a subtree is a count of the number of leaves that have a *good path* to the root of the subtree, where a good path is a path whose processing and network propagation times currently fall within the system's smallest supported TI bound TI_{min}. The difference $N_{all} - N_{reachable}$ thus represents the number of nodes whose inputs may fail to meet the system's tightest supported staleness bound; we will discuss what happens for attributes with TI bounds larger than TI_{min} momentarily.

Nodes compute $N_{reachable}$ in two steps:

1. *Basic aggregation*: PRISM creates a SUM aggregate and each leaf inserts local value of 1. The root of the tree then gets a count of all nodes.

2. *Aggressive pruning*: In contrast with the default behavior of retaining aggregate values of children as soft state for up to $T_{declareDead}$, $N_{reachable}$ must immediately change if the connection to a subtree is no longer a good path. Therefore, each internal node periodically probes each of its children. If a child c is not responsive, the node removes subtree c's contribution from the $N_{reachable}$ aggregate and immediately sends the new value up towards the root of the $N_{reachable}$ aggregation tree.

Temporal batching. If for an attribute the TI bound is relaxed to $TI_{attr} > TI_{min}$, PRISM uses the extra time $TI_{attr} - TI_{min}$ to batch updates for reducing load. To implement temporal batching, PRISM defines a narrow window of time during which a node must propagate updates to its parents; the details appear in an extended technical report [19]. However, an attribute's subtree that was unreachable over the last TI_{attr} could have been unlucky and missed its window even though it is currently reachable.

To avoid having to calculate a multitude of $N_{reachable}$ values for different TI bounds, PRISM modifies its temporal batching protocol to ensure that each attribute's promised TI bound is met for all nodes counted as reachable. In particular, when a node receives updates from a child marked unreachable, it knows those updates may be late and may have missed their propagation window. It therefore marks such updates as NODELAY. When a node receives a NODELAY update, it processes it immediately and propagates the result with the NODELAY flag so that temporal batching is temporarily suspended for that attribute. This modification may send extra messages in the (hopefully) uncommon case of a link performance failure and recovery, but it ensures that the current $N_{reachable}$ value counts nodes that are meeting all of their TI contracts.

4.2 Dynamic topology

Each virtual node in PRISM caches state from its children so that when a new input from one child comes in, it can use local information to compute new values to pass up. This information is soft state—a parent discards it if a child is unreachable for a long time. But because reconstructing this state is expensive (there may be tens of thousands of attributes for aggregation functions like "where is the nearest copy of file foo" [36]), PRISM uses long timeouts ($T_{declareDead} \approx 10$ minutes) to avoid spurious garbage collection.

As a result, when a subtree chooses a new parent, that subtree's inputs may still be stored by a former parent and thus may be counted multiple times in the aggregate as shown in Figure 5(d). N_{dup} exposes this inaccuracy by bounding the number of leaves whose inputs might be included multiple times in the aggregate query result. Note that PRISM allows a user to define duplicate-insensitive aggregation functions where possible [7, 24]. However, to support a broader range of aggregation functions, PRISM computes N_{dup} for each aggregation tree.

The basic aggregation function for N_{dup} is simple: if a subtree root spanning k leaf nodes switches to a new parent, that subtree root inserts the value k into the N_{dup} aggregate, which has SUM as its aggregation function. Later, when the node is certain that sufficient time has elapsed to ensure that its old parent has removed its soft state, it updates its input of N_{dup} to 0.

Our N_{dup} implementation must deal with two issues. First, for correctness, we must maintain the invariant that N_{dup} bounds the number of nodes whose inputs are double-counted despite failures and network delays. Second, for good performance, we must minimize the scope of disruptions when a tree reconfigures.

4.2.1 Lease aggregation

For correctness, PRISM ensures that N_{dup} always bounds the number of nodes with double-counted inputs despite network disruptions and delays (as opposed to

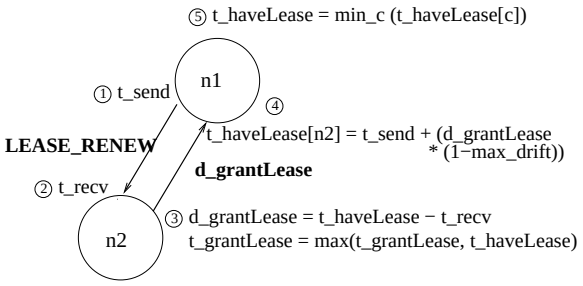
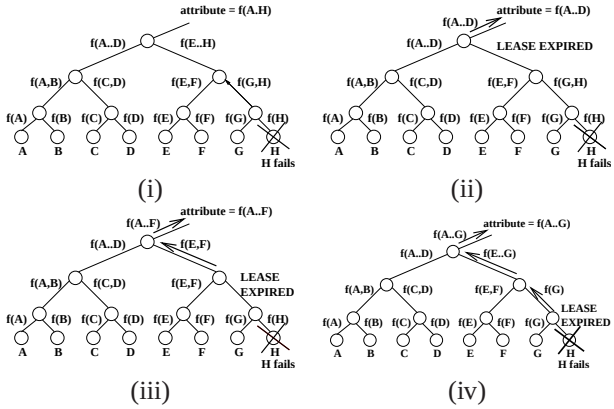
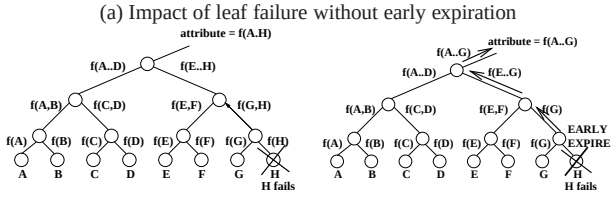


Figure 6: Protocol for a parent to renew a lease on the right to retain a child's contribution to an aggregate.



(a) Impact of leaf failure without early expiration



(b) Impact of leaf failure with early expiration

Figure 7: Recalculation of aggregate function across values A, B, ..., H after the node with input H fails (a) without and (b) with early expiration.

just providing a best-effort estimate). This guarantee is achieved through the use of a novel *lease aggregation* algorithm that extends the concept of leases [13] to hierarchical aggregation.

Figure 6 details the protocol used when a node n_1 updates a lease on the inputs from a set of descendants rooted at n_2. The algorithm makes use of local clocks at n_1 and n_2, but it is not sensitive to skew and tolerates a maximum drift rate of max_{drift} (e.g., 5%). In this protocol, a node maintains $t_{haveLease}$, the latest time for which it holds leases for all descendants, and $t_{grantLease}$, the latest time for which it has granted a lease to its ancestors. The key to the protocol is that the leases granted by a node are limited by the shortest lease held from any descendant.

Note that to cope with clock skew, the child n_2 extends the lease by a duration $d_{grantLease}$, but the child interprets the $d_{grantLease}$ interval starting from t_{recv}, the time it received the renewal request, while the parent interprets the interval starting from t_{send}. As a result, a lease always expires at a parent before expiring at any descendent regardless of the skew between their clocks [42].

A node that roots a k-leaf subtree that switches to a new parent then contributes k to N_{dup} until $t_{grantLease}$, after which it may reset its contribution of N_{dup} to 0 because its former parent is guaranteed to have cleared from its soft state all inputs from that node.

4.2.2 Early expiration

For good performance, PRISM uses *early expiration* to minimize the scope of disruption when a tree's topology reconfigures. In particular, the lease aggregation mechanism ensures the invariant that leases near the root of a tree are shorter than leases near the leaves. As a result, a naive implementation that removes cached soft state exactly when a lease expires would exhibit the perverse behavior illustrated in Figure 7(a): each node from the root to the parent of a failed node will successively expire its problematic child's state, recalculate its aggregates without that child, update its parent, renew its parent's lease, and then repeatedly receive and propagate updated aggregates from its child as the process ripples down the tree. Not only is that process expensive, but it may significantly and unnecessarily perturb values reported at the root for all attributes by removing and readding large subtrees of inputs. For example, in Figure 7-ii, the leaf failure at node H temporarily removes inputs E, F, and G from the aggregate. Furthermore, note that the example in Figure 7 is a common case: in a randomly constructed tree, the vast majority of nodes are near the leaves. Failing to address this problem would transform the common-case of leaf failures into significant disruptions and bring into play the amplification effect.

Early expiration avoids this unwarranted disruption as Figure 7(b) illustrates. A node at level i of the tree discards the state of an unresponsive subtree ($maxLevels$ - i) * d_{early} before its lease expires. Once the node has removed the problematic child's inputs from the aggregates values it has reported to its parent, the node can renew leases to its parent that are no longer limited by the ever-shortening lease held on the problematic child. As the figure illustrates, this technique minimizes disruption by allowing a node near the trouble spot to prune the tree, update its ancestors, and resume granting long leases *before* any ancestor acts.

4.3 Scaling to large systems

Scaling NI is a challenge. To scale attribute monitoring to large numbers of nodes and attributes, PRISM constructs a forest of trees using an underlying DHT and then uses different aggregation trees for different attributes [40]. As Figure 8 illustrates, a failure affects different

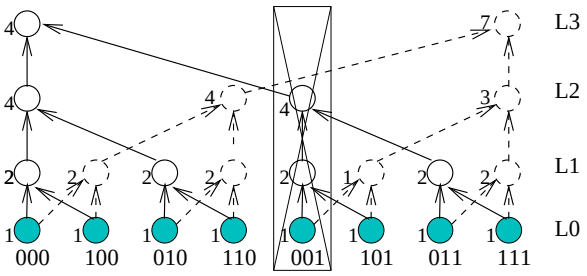


Figure 8: The failure of a physical node has different effects on different aggregations depending on which virtual nodes are mapped to the failed physical node. The numbers next to virtual nodes show the value of $N_{reachable}$ for each subtree after the failure of physical node 001, which acts as a leaf for one tree but as a level-2 subtree root for another.

trees differently. The figure shows 2 aggregation trees corresponding to keys 000 and 111 for a 8-node system. In this system, the failure of the physical node with key 001 removes only a leaf node from the tree 111 but disconnects a 2-level subtree from the tree 000 highlighting the amplification effect. Therefore, quantifying the effect of failures will require calculating NI metrics for each of the n distinct global trees in an n-node system. Making matters worse, as Section 4.1 explained, maintaining the NI metrics requires frequent active probing along each edge in each tree.

As a result of these factors, the straightforward algorithm for maintaining NI metrics separately for each tree is not tenable: the DHT forest of n degree-d aggregation trees with n physical nodes and each tree having $\frac{n-1/d}{1-1/d}$ edges ($d > 1$), has $\Theta(n^2)$ edges that must be monitored; such monitoring would require $\Theta(n)$ messages per node every probe interval (p = 10s in PRISM prototype). To put this in perspective, consider a n=1024-node system with d=16-ary trees (i.e., a DHT with 4-bit correction per hop). The straightforward algorithm then has each node sending over roughly 100 probes per second. As the system grows, the situation deteriorates rapidly—a 16K-node system requires each node to send roughly 1600 probes per second.

Our solution, described below, reduces active monitoring work to $\Theta(d \log_d n)$ probes per node per p seconds. The 1024-node system in the example would require each node to send about 5 probes per second; the 16K-node system would require each node to send about 7 probes per second.

4.3.1 *Dual tree prefix aggregation*

To make it practical to maintain the NI values, we take advantage of the underlying structure of our Plaxton-tree-based DHT [28] to reuse common subcalculations across different aggregation trees using a novel *dual tree prefix aggregation* abstraction.

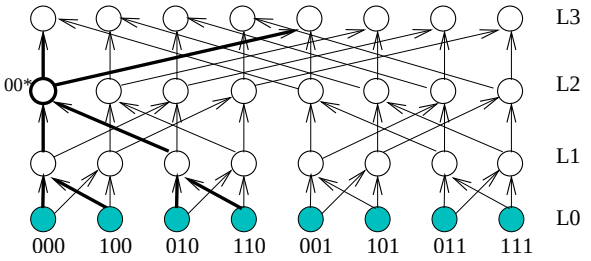


Figure 9: Plaxton tree topology is an approximate butterfly network. The bold connections illustrate how a virtual node 00* uses the dual tree prefix aggregation abstraction to aggregate values from a tree below it and distribute the results up a tree above it.

As Figure 9 illustrates, this DHT construction forms an approximate butterfly network. For a degree-d tree, the virtual node at level i has an id that matches the keys that it routes in $\log d * i$ bits. It is the root of exactly one tree, and its children are approximately d virtual nodes that match keys in $\log d * (i-1)$ bits. It has d parents, each of which matches different subsets of keys in $\log d * (i+1)$ bits. But notice that for each of these parents, this tree aggregates inputs from *the same subtrees*.

Whereas the standard aggregation abstraction computes a function across a set of subtrees and propagates it to one parent, a *dual tree prefix aggregation* computes an aggregation function across a set of subtrees and propagates it to *all parents*. As Figure 9 illustrates, each node in a dual tree prefix aggregation is the root of two trees: an aggregation tree below that computes an aggregation function across a set of leaves and a distribution tree above that propagates the result of this computation to a collection of enclosing aggregates that depend on this sub-tree for input.

For example in Figure 9, consider the level 2 virtual node 00* mapped to node 000. This node's $N_{reachable}$ count of 4 represents the total number of leaves included in that virtual node's subtree. This node aggregates this single $N_{reachable}$ count from its descendants and propagates this value to both of its level-3 parents, 000 and 001. For simplicity, the figure shows a binary tree; by default PRISM corrects 4 bits per hop and d=16, so each subtree is common to 16 parents.

5. CASE-STUDY APPLICATIONS

We have developed a prototype of the PRISM monitoring system on top of FreePastry [30]. To guide the system development and to drive the performance evaluation, we have also built three case-study applications using PRISM: (1) a distributed heavy hitter detection service, (2) a distributed monitoring service for Internet-scale systems, and (3) a distributed bot detector service.

Distributed Heavy Hitter detection (DHH). Our first application is identifying heavy hitters in a distributed

system—for example, the top 10 IPs that account for a significant fraction of total incoming traffic in the last 10 minutes [8, 20]. The key challenge for this distributed query is scalability for aggregating per-flow statistics for tens of thousands to millions of concurrent flows in real-time. For example, a subset of the Abilene [1] traces used in our experiments include 80 thousand flows that send about 25 million updates per hour.

To scalably compute the global heavy hitters list, we chain two aggregations where the results from the first feed into the second. First, PRISM calculates the total incoming traffic for each destination address from all nodes in the system using SUM as the aggregation function and hash(HH-Step1, destIP) as the key. For example, tuple (H = hash(HH-Step1, 128.82.121.7), 700 KB) at the root of the aggregation tree T_H indicates that a total of 700 KB of data was received for 128.82.121.7 across all vantage points during the last time window. In the second step, we feed these aggregated total bandwidths for each destination IP into a SELECT-TOP-10 aggregation with key hash(HH-Step2, TOP-10) to identify the TOP-10 heavy hitters among all flows.

PrMon. The second case-study application is PrMon, a distributed monitoring service that is representative of monitoring Internet-scale systems such as PlanetLab [27] and Grid systems [37] that provide platforms for developing, deploying, and hosting global-scale services. For instance, to manage a wide array of user services running on the PlanetLab testbed, the system administrators need a global view of the system to identify problematic services (slices in PlanetLab terminology) e.g., if any slice is consuming more than 10GB of memory across all nodes on which it is running. Similarly, users require system state information to query for "lightly-loaded" nodes for deploying new experiments or to track the resource consumption of their running experiments.

To provide such information in a scalable way and in real-time, PRISM computes the per-slice aggregates for each resource attribute (e.g., CPU, MEM, etc.) along different aggregation trees. This aggregate usage of each slice across all PlanetLab nodes for a given resource attribute (e.g., CPU) is then input to a per-resource SELECT-TOP-100 aggregate (e.g., SELECT-TOP-100, CPU) to compute the list of top-100 slices in terms of consumption of the resource.

PrBot. The final monitoring application is PrBot, a distributed bot detector service to keep track of which nodes are contacting a large number of other nodes. In this application, a leaf sensor at each node maintains a sketch [7] data structure to count the number of distinct elements in the set of destination IP addresses to which that node has sent a packet. The sketch outputs a cardinality estimate of this set which is fed into PRISM as (PrBot, Sensor-Id, set-size) to compute a top-100 list of nodes that might be used as bots. In the presence of failures, our aim is to accurately compute this list so as not to raise any false alarms (false positives) or miss any inputs (false negatives).

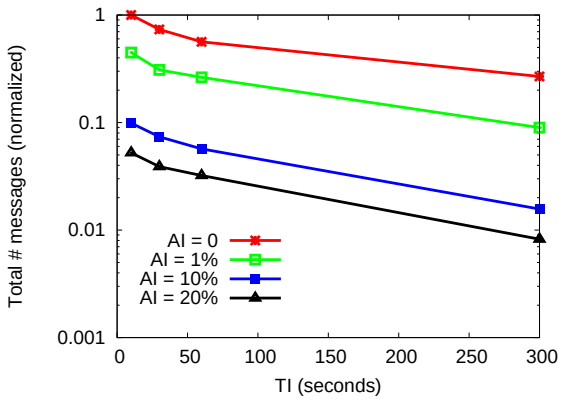


Figure 10: Load vs. AI and TI for DHH application.

6. EXPERIMENTAL EVALUATION

Our experiments focus on investigating the consistency-availability trade-offs that NI exposes, and quantifying the overhead in computing the NI metrics. Overall, our evaluation shows that PRISM is an effective substrate for accurate scalable monitoring: the NI metrics both successfully characterize system state and reduce measurement inaccuracy while incurring a small communication overhead.

We run our experiments in controlled environments (clusters of 100 Emulab [39] or 20 department Condor machines), and PlanetLab [27] up to 96 nodes.

6.1 Scalability benefits

First, we quantify the scalability benefits from PRISM's combination of hierarchical aggregation, arithmetic filtering, and temporal batching for the DHH application. We use multiple netflow traces obtained from the Abilene [1] Internet2 backbone network. Figure 10 shows the precision-performance results running DHH on 400 nodes mapped to 100 physical Emulab machines; the total monitoring load is normalized relative to the load for AI of 0 and TI of 10 seconds. Note that AI and TI each reduce monitoring overheads by nearly an order of magnitude. We examine the PrMon and the PrBot applications in the extended technical report and observe similar results [19].

6.2 NI: Exposing disruption

In this section, we analyze the effectiveness of NI metrics in reflecting network state and filtering inaccurate reports.

We first illustrate how NI metrics reflect network state for a small scale controlled experiment. In Figure 11, we run a 20 node experiment on the departmental Condor cluster where we kill a single node at t = 815 seconds

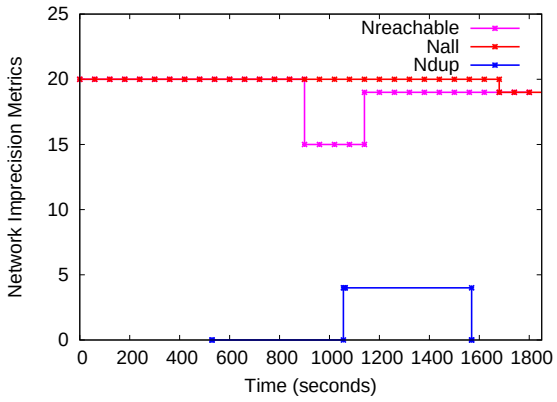


Figure 11: NI metrics under induced system churn – single node failure at 815 seconds into the experimental run.

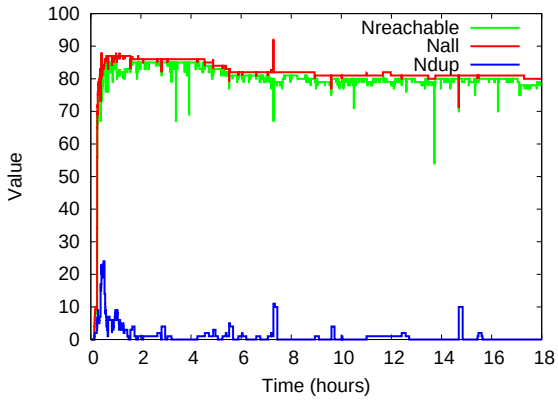


Figure 12: NI metrics reflecting PlanetLab state (85 nodes).

into the run and observe the variation of reported NI metrics for an attribute with TI of 60 seconds. This failure causes the $N_{reachable}$ value to fall from 20 to 15 within 40 seconds after the node failure. The drop in $N_{reachable}$ indicates that any result calculated in this interval might only include correct values from 15 nodes. After about 240 seconds, the underlying DHT declares the missing node to be dead and reconfigures the topology. Correspondingly, the N_{dup} value goes from 0 to 4 at about t = 1060 seconds when the disconnected nodes join new parents and start reporting their N_{dup} value. The N_{all} value remains stable from 20 until about t=1600 seconds to reflect the long $T_{declareDead}$ timeout from the failure at t = 815 seconds before the system declares unreachable nodes to be dead. Finally, the N_{dup} value falls back to 0 and both N_{all} and $N_{reachable}$ stabilize at 19 (nodes) denoting that the system is back to a stable state.

For subsequent experiments, we focus on NI's effectiveness during periods of instability. In particular, we run experiments on PlanetLab nodes. Because these nodes

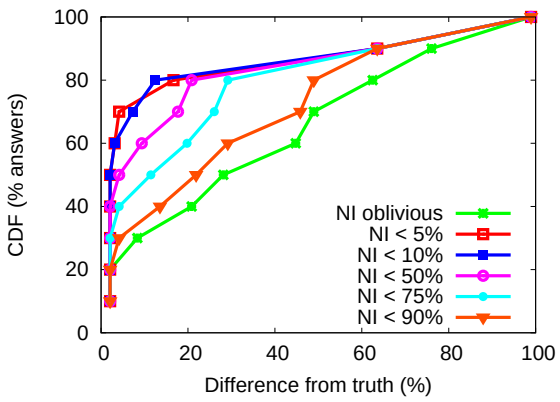


Figure 13: CDF for reported answers filtered for different NI thresholds and k = 1.

show heavy load, unexpected delays, and relatively frequent reboots (especially prior to deadlines!), we expect these nodes to exhibit more NI than in a typical distributed environment, which makes them a convenient stress test of our system.

Figure 12 shows how NI reflects network state for a 85-node PrMon experiment on PlanetLab for an 18-hour run. We observe that even without any induced failures, there are short-term instabilities in values reported by $N_{reachable}$, N_{all}, and N_{dup} due to missing/delayed ping reply messages for $N_{reachable}$ and lease expirations triggered by DHT reconfigurations for N_{dup}. During the course of the run, 5 of the 85 nodes became unresponsive; hence the final $N_{reachable}$ and N_{all} values stabilize at 80.

6.3 Coping with disruption

Next we quantify the risks of reporting global aggregate results without incorporating NI. We run a 1 hour PrMon experiment on 94 PlanetLab nodes for an attribute computing a SUM aggregate with AI = 0 and TI = 10 seconds. Here we present the results for the SUM aggregate since it is common in our three case-study and several other applications; the results for other standard aggregation functions (e.g., MAX, MIN, AVG, etc.) are described in an extended technical report [19]. Figure 13 shows the CDF of reported answers showing the deviation in reports with respect to an oracle that has instantaneous access to all inputs; we simulate this oracle via off-line processing of input logs. The different lines in the graph correspond to the reported answers filtered for different NI thresholds. For simplicity, we condense NI to a single parameter MAX$[\frac{N_{all}-N_{reachable}}{N_{all}}, \frac{N_{dup}}{N_{all}}]$. We observe that NI effectively reflects the stability of network state: when NI $<$ 5%, 80% answers have less than 20% deviation from the true value. Conversely, for monitoring systems that ignore NI (the *NI oblivious* line), half of their reports differ from the truth by more than 60%.

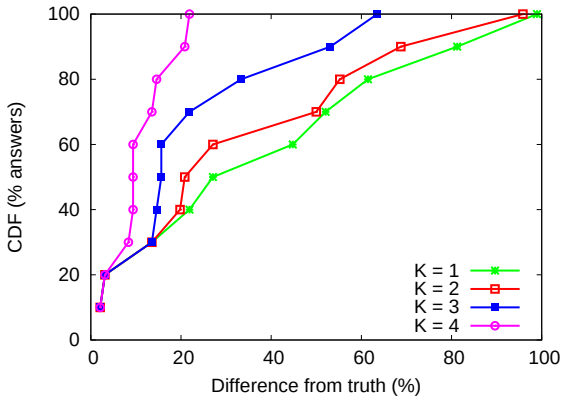


Figure 14: CDF of NI values for different k.

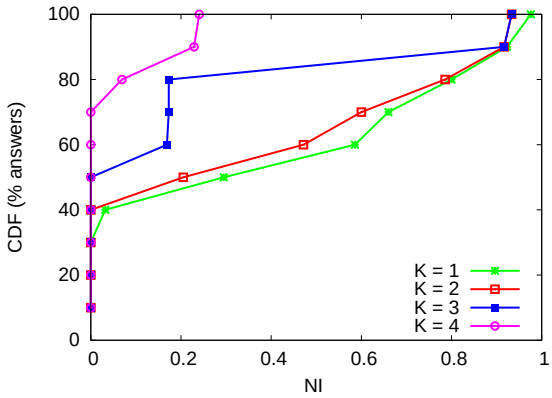


Figure 15: CDF of NI values for k duplicate keys.

As discussed in Section 3.3, applications can filter results using different NI thresholds and take an appropriate action to correct distorted results.

In Figure 14 we explore the effectiveness of the redundant aggregation approach discussed in Section 3.3 i.e., using k redundant trees to compute an attribute SUM and then using NI to identify the highest-quality result. Figure 14 shows the CDF of results with respect to the deviation from an oracle as we vary k from 1 to 4. When deviation is less than 10% (small NI), retrieving results from the root of one aggregation tree (k = 1) suffices. However, for large deviation, fetching the reports from only one aggregation tree can introduce deviation as high as 100% whereas choosing the result from the most stable of 4 trees reduces the deviation to at most 22% thereby reducing the worst-case inaccuracy by nearly a factor of 5. Note that PRISM enables a trade-off: for a given bandwidth budget, a system may be able to use small increases in arithmetic filtering and temporal batching to increase k and thereby greatly reduce NI.

Filtering answers during periods of high churn exposes a fundamental consistency versus availability tradeoff [12]. Figure 15 shows how varying k allows us to increase monitoring load to improve this tradeoff. As k increases, the fraction of time during which NI is low increases. The intuition is that because the vast majority of nodes

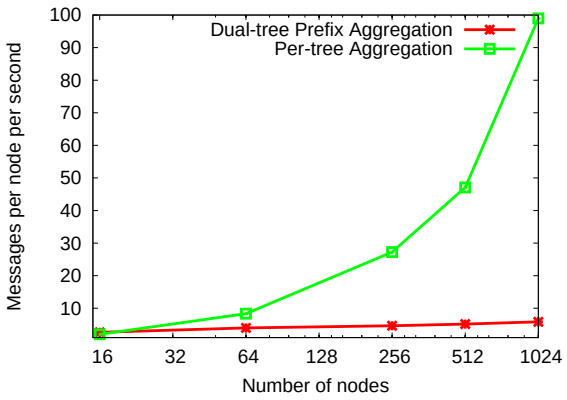


Figure 16: NI monitoring overhead for dual-tree prefix aggregation compared to computing NI per aggregation tree; x-axis is on a log scale.

in any 8-ary tree are near the leaves, sampling several trees rapidly increases the probability that at least one tree avoids encountering many near-root failures. We provide an analytic model formalizing this intuition in our technical report [19].

6.4 NI scalability

Finally, we quantify the monitoring overhead of tracking NI via (1) each aggregation tree and (2) dual-tree prefix aggregation. Figure 16 shows the average per-node message cost for NI monitoring varying network size from 16 to 1024 nodes. We observe that the overhead using per aggregation tree scales linearly with the network size whereas it scales logarithmically using dual-tree prefix aggregation.

Note that the above experiment constructs all n trees in the DHT forest of n nodes assuming that the number of attributes is at least the number of nodes n. However, for systems that aggregate fewer attributes, it is important to know which of the above two techniques for tracking NI is more efficient. Figure 17 shows both the average and the maximum message cost across all nodes in a 1000-node experiment for both per-tree NI aggregation and dual-tree prefix aggregation. We observe that the break-even point for the average load is 44 trees (4.4%) while the break-even point for the maximum load is only 8 trees (0.8%).

7. RELATED WORK

The idea of flagging results when the state of a distributed system is disrupted by node or network failures has been used in tackling other distributed systems problems. For example, our idea of NI is analogous to that of failure detectors [5] for fault-tolerant distributed systems. Freedman et al. propose link-attestation groups abstraction in [11] that uses an application specific notion of reliability and correctness, so as to map which pairs of nodes consider each other reliable. Their system,

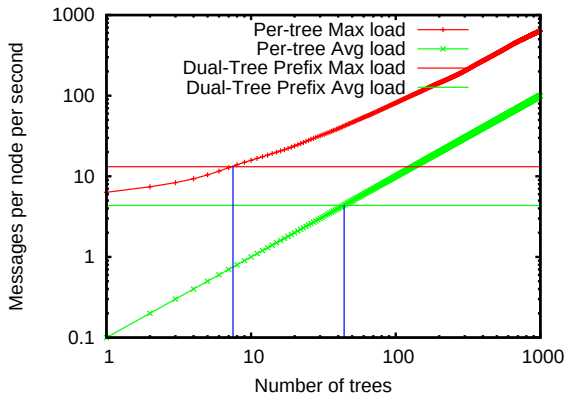


Figure 17: Comparing NI tracking overhead by varying the number of trees (attributes) for (a) per-tree aggregation vs. (b) dual-tree prefix aggregation in a 1000-node system. The figure shows both the AVG load and the MAX load, along with the break-even points.

designed for groups on the scale of tens of nodes, monitors the nodes and system and exposes such attestation graph to the applications. Bawa et. al [3] survey previous work on measuring the validity of query results in faulty networks. Their "single-site validity" semantic is equivalent to PRISM's $N_{reachable}$ metric. *Completeness* [14] defined as the percentage of network hosts whose data contributed to the final query result, is similar to the ratio of $N_{reachable}$ and N_{all}. Relative Error [7, 43] between the reported and the "true" result at any instant can only be computed by an oracle with a perfect view of the dynamic network.

Several aggregation systems have worked to address the failure amplification effect. To mask failures, TAG [21] proposes (1) reusing previously cached values and (2) dividing the aggregate value into fractions equal to the number of parents and then sending each fraction to a distinct parent. This approach only reduces the variance but not the expected value of the aggregate value at the root. SAAR uses multiple interior-node-disjoint trees to reduce the impact of node failures [23]. Other studies have proposed multi-path routing methods [7, 14, 18, 22, 24] for fault-tolerant aggregation.

Recent proposals [3, 7, 22, 24, 44] have combined multipath routing with order- and duplicate-insensitive data structures to tolerate faults in sensor network aggregation. The key idea is to use probabilistic counting [9] to approximately count the number of distinct elements in a multi-set. PRISM takes a complementary approach: whereas multipath duplicate-insensitive (MDI) aggregation seeks to reduce the effects of network disruption, PRISM's NI metric seeks to quantify the network disruptions that do occur. In particular, although MDI aggregation can, in principle, reduce network-induced inaccuracy to any desired target if losses are independent and sufficient redundant transmissions are made [24], the systems studied in the literature are still subject to non-zero network-induced inaccuracy due to efforts to balance transmission overhead with loss rates, insufficient redundancy in a topology to meet desired path redundancy, or correlated network losses across multiple links. These issues may be more severe in our environment than in wireless sensor networks targeted by MDI approaches because the dominant loss model may differ (e.g., link congestion and DHT reconfigurations in our environment versus distance-sensitive loss probability for the wireless sensors) and because the transmission cost model differs (for some wireless networks, transmission to multiple destinations can be accomplished with a single broadcast.)

The MDI aggregation techniques are also complementary in that PRISM's infrastructure provides NI information that is common across attributes while the MDI approach modifies the computation of individual attributes. As Section 3.3 discussed, NI provides a basis for integrating a broad range of techniques for coping with network error, and MDI aggregation may be a useful technique in cases when (a) an aggregation function can be recast to be order- and duplicate-insensitive and (b) the system is willing to pay the extra network cost to transmit each attribute's updates. Further, to realize this promise, additional work is required to extend MDI approach to bounding the approximation error while still minimizing network load via AI and TI filtering.

Consistency has long been studied in the context of non-aggregating file systems and databases. Yu et al. [45] propose three metrics—Numerical Error, Order Error, and Staleness—to capture the consistency spectrum in a distributed replicated system where any node can perform read or write operations. Numerical error is similar to AI and Staleness is similar to TI. Payton et. al [26] proposed a query processing model for one-shot, non-aggregate queries in mobile ad hoc and sensor networks.

Consistency for aggregation, however, is fundamentally different. For example, aggregation systems are large-scale with many concurrent writers which implies that it is not feasible to resolve CAP dilemma [12] by blocking reads during periods when a writer may be disconnected. So we emphasize availability by providing conditional consistency: operations always complete but results are annotated with information about their quality.

8. CONCLUSIONS

> If a man will begin with certainties, he shall end in doubts: but if he will be content to begin with doubts, he shall end in certainties.
> –Sir Francis Bacon

We have presented Network Imprecision, a new metric of characterizing network state that quantifies the consistency of aggregate query results in a dynamic large-scale

monitoring system. Without NI guarantees, large scale network monitoring systems may provide misleading reports because query result outputs by such systems may be arbitrarily wrong. Incorporating NI in the PRISM monitoring framework qualitatively improves its output by exposing cases when approximation bounds on query results can not be trusted.

9. REFERENCES

[1] Abilene internet2 network. `http://abilene.internet2.edu/`.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[3] M. Bawa, A. Gionis, H. Garcia-Molina, and R. Motwani. The price of validity in dynamic networks. In *SIGMOD*, 2004.

[4] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. In *SIGCOMM*, Portland, OR, August 2004.

[5] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, Mar. 1996.

[6] D. D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A knowledge plane for the internet. In *SIGCOMM*, 2003.

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

[8] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, 2002.

[9] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences,*, 31(2):182–209, Oct. 1985.

[10] M. J. Freedman and D. Mazires. Sloppy Hashing and Self-Organizing Clusters. In *IPTPS*, Berkeley, CA, February 2003.

[11] M. J. Freedman, I. Stoica, D. Mazieres, and S. Shenker. Group therapy for systems: Using link attestations to manage failures. In *IPTPS*, 2006.

[12] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, Jun 2002.

[13] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *SOSP*, pages 202–210, 1989.

[14] I. Gupta, R. van Renesse, and K. P. Birman. Scalable fault-tolerant aggregation in large process groups. In *DSN*, 2001.

[15] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USITS*, March 2003.

[16] J. M. Hellerstein, V. Paxson, L. L. Peterson, T. Roscoe, S. Shenker, and D. Wetherall. The network oracle. *IEEE Data Eng. Bull.*, 28(1):3–10, 2005.

[17] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *VLDB*, 2003.

[18] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom*, 2000.

[19] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. Technical Report TR-06-22, UT Austin Department of Computer Sciences, March 2006.

[20] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. Star: Self tuning aggregation for scalable monitoring. In *33rd International Conference on Very Large Databases (VLDB)*, 2007.

[21] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *OSDI*, 2002.

[22] A. Manjhi, S. Nath, and P. B. Gibbons. Tributaries and deltas: efficient and robust aggregation in sensor network streams. In *SIGMOD*, 2005.

[23] A. Nandi, A. Ganjam, P. Druschel, T. S. E. Ng, I. Stoica, H. Zhang, and B. Bhattacharjee. Saar: A shared control plane for overlay multicast. In *NSDI*, 2007.

[24] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys*, 2004.

[25] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[26] J. Payton, C. Julien, and G.-C. Roman. Automatic consistency assessment for query results in dynamic environments. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2007. (to appear).

[27] Planetlab. `http://www.planet-lab.org`.

[28] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *ACM SPAA*, 1997.

[29] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In *SIGCOMM*, 2001.

[30] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-peer Systems. In *Middleware*, 2001.

[31] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An infrastructure for connecting sensor networks and applications. Technical Report TR-21-04, Harvard Technical Report, 2004.

[32] A. Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell, 1992.

[33] A. Silberstein, G. Puggioni, A. Gelfand, K. Munagala, and J. Yang. Suppression and failures in sensor networks: A bayesian approach. In *33rd International Conference on Very Large Databases (VLDB)*, 2007.

[34] A. Singla, U. Ramachandran, and J. Hodgins. Temporal notions of synchronization and consistency in Beehive. In *Proc. SPAA*, 1997.

[35] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[36] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Design Considerations for Distributed Caching on the Internet. In *ICDCS*, May 1999.

[37] `http://www.globus.org/`.

[38] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *TOCS*, 21(2):164–206, 2003.

[39] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[40] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc SIGCOMM*, Aug. 2004.

[41] P. Yalagandula, P. Sharma, S. Banerjee, S.-J. Lee, and S. Basu. S^3: A Scalable Sensing Service for Monitoring Large Networked Systems. In *Proceedings of the SIGCOMM Workshop on Internet Network Management*, 2006.

[42] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proc USITS*, Oct. 1999.

[43] R. G. Yonggang Jerry Zhao and D. Estrin. Computing aggregates for monitoring wireless sensor networks. In *SNPA*, 2003.

[44] H. Yu. Dos-resilient secure aggregation queries in sensor networks. In *PODC*, pages 394–395, 2007.

[45] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. on Computer Systems*, 20(3), Aug. 2002.

[46] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical Report UCB/CSD-01-1141, UC Berkeley, Apr. 2001.