PRISM: PRecision-Integrated Scalable 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 describes PRISM, a scalable monitoring service that makes *imprecision* a first-class abstraction for its scalable DHT-based aggregation service. Exposing imprecision is essential for both correctness in the face of network and node failures and scalability to large systems. PRISM introduces the notion of *conditioned consistency* that quantifies imprecision along a three-dimensional vector: *arithmetic imprecision* (AI) bounds numeric inaccuracy, *temporal imprecision* (TI) bounds update delays, and *network imprecision* (NI) bounds uncertainty due to network and node failures. AI and TI balance precision against monitoring overhead for scalability while NI addresses the fundamental challenge of providing consistency guarantees despite failures in a large distributed system. Our implementation addresses the challenge of providing these metrics while scaling to a large numbers of nodes and attributes. By introducing a 10% AI, PRISM's PlanetLab monitoring service, PrMon, can reduce network overheads by an order of magnitude compared to the currently-used CoMon service. And, 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.

1 Introduction

This paper describes how to make *imprecision* a first class abstraction for large-scale system monitoring.

Scalable system monitoring is a fundamental abstraction for large-scale networked systems, and it can serve as a basic building block for new applications such as network monitoring and management [5, 15, 41], resource location [16, 40], efficient multicast [36], sensor networks [16, 40], resource management [40], and bandwidth provisioning [9]. Recent work on aggregation [16, 30, 36, 40] and DHTs [27–29, 33, 46] provides important enabling technology for constructing monitoring systems that are self-organizing, scalable, and robust [36, 40].

However, to realize this vision of scalable system monitoring, the underlying monitoring infrastructure must expose imprecision in a controlled manner for two reasons.

First, correct interpretation of data requires explicitly exposing the imprecision introduced by sensor inaccuracy and node/network delays and failures. A fundamental and unique challenge in any hierarchical aggregation system is the *failure amplification effect*: if a non-leaf node fails, an entire subtree rooted at that node is affected. For example, failure of a level-3 node in a degree-8 aggregation tree can cut off updates from 512 leaf nodes. As a result, a hierarchical monitoring service that does not expose imprecision risks delivering arbitrarily incorrect results.

Second, introducing controlled amounts of imprecision can reduce monitoring load by an order of magnitude or more for some applications. Studies suggest [20, 22, 32, 36, 44] that real-world applications often can tolerate some inaccuracy as long as the maximum error is bounded and small amounts of imprecision can provide significant bandwidth reductions. This enables new classes of precision-aware monitoring applications that can tradeoff between imprecision and resource usage.

To meet these needs, we have developed PRecision-Integrated Scalable Monitoring (PRISM). The PRISM system makes two contributions: First, it defines a novel *conditioned consistency* metric that quantifies imprecision along a three-dimensional vector: **(Arithmetic, Temporal, Network)**.

- *Arithmetic* imprecision (AI) bounds the numerical inconsistency between the reported value of an aggregate relative to the true value.
- *Temporal* imprecision (TI) places a real-time bound on the delay from when an event/update occurs until it is reported.
- *Network* imprecision (NI) bounds the inaccuracy introduced by failed/slow nodes, failed/slow network links, and aggregation tree reconfigurations.

Although each of the three dimensions is individually useful, the combination is vital because it enables *conditioned consistency*: the arithmetic and temporal guarantees are calculated optimistically, assuming that the network is "well behaved" (e.g., no node failures, slow links, or tree reconfigurations have affected the results). The NI metric then qualifies AI and TI metrics by quantifying how "well behaved" the network actually has been during the period when these metrics are calculated.

Second, it provides a scalable implementation of each of these three metrics for DHT-based aggregation systems. Scalability to large numbers of attributes and nodes is vital because network monitoring applications may track tens of thousands of attributes across hundreds or thousands of nodes [34, 36, 40].

- For AI, the challenge is distributing an imprecision budget across nodes based on each attribute's workload. PRISM employs a hierarchical self-tuning algorithm that directs imprecision slack to where it is most needed and that tries to ensure that the adaptation cost is smaller than the benefits of doing so.

- For TI, the challenge is to maximize the number of updates batched together and to minimize the TI introduced by this batching. To accomplish this goal, PRISM pipelines the available slack across levels of the aggregation hierarchy.
- For NI, the challenge is to scalably detect and report failed/slow nodes/links which requires active probing. A straightforward algorithm that detects and aggregates NI values along each aggregation tree in an n-node system can lead to $O(n)$ message load at each node in every probing period. By leveraging the observation that the forest of aggregation trees forms a *butterfly network*, PRISM introduces a novel *dual-tree prefix aggregation abstraction* that re-uses work done by subtrees and thereby reduces the per-node cost to $O(\log n)$ messages every probing period. For a 1000-node system, this implies three orders of magnitude reduction in message cost compared to the naive algorithm above.

Experience with a distributed heavy hitter detection application and a PrMon monitoring service for PlanetLab built on PRISM illustrate how explicitly managing imprecision can qualitatively enhance a monitoring service. The most obvious benefit is improved scalability: for both applications, small amounts of imprecision drastically reduce monitoring load or allow more extensive monitoring for a given load budget. For example, in PrMon, a 10% AI allows us to reduce network load by an order of magnitude compared to the widely used CoMon [6] service. A subtler but perhaps more important benefit is the ability to quantify and improve confidence in the accuracy of outputs by addressing network imprecision and the amplification effect. For example, 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.

The key contributions of this paper are as follows. First, we present PRISM, the first DHT-based system that enables imprecision for scalable aggregation by introducing a new conditioned consistency metric that bounds the arithmetic, temporal, and network imprecision. Second, we provide scalable and efficient implementation of each precision metric via (1) self-tuning of AI budgets, (2) pipelining of TI delays, and (3) dual-tree prefix aggregation for NI. Third, our evaluation demonstrates that imprecision is vital for enabling scalable aggregation: a system that ignores imprecision can silently report arbitrarily incorrect results and a system that fails to exploit imprecision can impose unacceptable overheads.

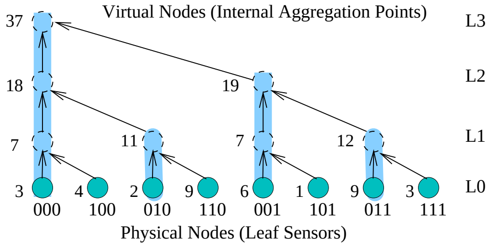

Virtual Nodes (Internal Aggregation Points)
L3
L2
L1
L0
37
18
19
7
11
7
12
3
4
2
9
6
1
9
3
000
100
010
110
001
101
011
111
Physical Nodes (Leaf Sensors)

Fig. 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

PRISM builds on two recent and ongoing research efforts for scalable monitoring: aggregation [36] and DHT-based aggregation [40].

Aggregation. Aggregation is a fundamental abstraction for scalable monitoring [10, 16, 27, 36, 40] because it allows applications to access summary views of global information and detailed views of rare events and nearby information.

The aggregation abstraction in PRISM is defined across 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. 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). 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.

The tree-based aggregation in the PRISM framework is defined in terms of an aggregation function which is installed at all the nodes in the tree. Each leaf node (physical sensor) inserts or modifies its local value for an attribute defined as an {attribute type, attribute name} pair which is recursively aggregated up the tree. For each level-i subtree T_i in the aggregation tree, PRISM defines an *aggregate value* $V_{i,attr}$ for each attribute as follows: For a (physical) leaf node T_0 at level 0, $V_{0,attr}$ is the locally stored value for the attribute or NULL if no matching tuple exists. Then the aggregate value for a level-i subtree T_i is the aggregation function for the attribute type, A_{type}, computed across the aggregate values of each of T_i's k children. Figure 1, for example, illustrates the computation of a simple SUM aggregate.

DHT-based aggregation. To achieve scalability for Internet-scale systems, PRISM faces the fundamental challenge of computing aggregates for thousands to millions of attributes across hundreds or thousands of nodes [36, 40]. Later in this section, we present an example of detecting heavy hitters on a distributed system where PRISM needs to track millions of attributes. To address this scalability challenge, PRISM leverages DHTs [27–29, 33, 46] to construct a forest of aggregation trees and maps different attributes to different trees [40]. 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 along the aggregation tree corresponding to *DHTtree*$_k$, different attributes are load balanced across different trees.

Example Applications Aggregation is a building block for many distributed applications such as network management [41], service placement [12], sensor monitoring and control [20], multicast tree construction [36], and naming and request routing [7]. In this paper, we focus on two case-study examples: a distributed heavy hitter detection and PrMon, a distributed monitoring service for PlanetLab modelled on CoMon [6].

Heavy Hitter detection: Our first application is identifying heavy hitters on a distributed system i.e., the top 10 IPs that account for a significant fraction of total incoming traffic in a measurement interval (e.g., 10 minutes) [9]. The key challenge for this distributed query is scalability for aggregating per-flow statistics for millions of concurrent flows in real-time; the Abilene [1] traces used in our experiments include up to 3.4 million flows per hour.

To scalably compute the global heavy hitters list, we chain two aggregations where the results from the first aggregation feed into the second aggregation. In the first aggregation, PRISM calculates the total bandwidth consumed by each sender to all nodes in the system using SUM as the aggregation function and {HH-Step1,senderIP} as the key. For example, a node writes the tuple ({HH-Step1, 128.82.121.7}, 700 KB) indicating that 700 KB of data was received from the node 128.82.121.7 during the last time window. In the second step, we feed these aggregated total bandwidths for each sender IP address into another aggregation tree for selecting TOP-10 heavy hitters. To achieve this, we use SELECT-TOP-10 as the aggregation function and use {HH-Step2,TOP-10} as the key. For example, the root of the first aggregation tree for {HH-Step1,128.82.121.7} which has computed the global aggregate value of 6200KB as the total bandwidth consumed by 128.82.121.7, inputs the tuple ({HH-Step2, TOP-10},{128.82.121.7,6200 KB}). At the end of this chained aggregation, the root of the second aggregation tree has the top 10 IP addresses that send most traffic to the nodes in the system.

Real-time Network Monitoring: The second application is our PrMon monitoring service that is representative of monitoring Internet-scale distributed systems such as PlanetLab [26] and Grid systems [35] that provide open 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 experiments (slices in PlanetLab terminology) to identify, for example, any slice consuming more than 500GB 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, TX1, 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. Although there are existing central monitoring services, in Section 5 we will show that PRISM can monitor a large number of attributes at much finer time scales while incurring significantly lower network costs.

3 AI and TI

PRISM quantifies imprecision along a three-dimensional vector: (Arithmetic, Temporal, Network). We now describe how we enforce bounds on *arithmetic imprecision* (AI), which limits the numeric difference between a reported value of an attribute and its true value [23, 45], and *temporal imprecision* (TI), which limits the delay from when an update is input at a leaf sensor until the effects of the update are reflected in the root aggregate. These aspects of imprecision provide means to (a) expose inherent imprecision in a monitoring system stemming from sensor inaccuracy and update propagation delays and (b) reduce system load by introducing additional filtering and batching on update propagation.

The implementations of AI and TI are simple because they can assume that aggregation trees never reconfigure and that nodes and network paths never fail and are never slow. The *network imprecision* (NI) metric described in Section 4 addresses these challenging real-world issues.

3.1 Arithmetic Imprecision (AI)

We first describe the basic mechanism for enforcing AI for each aggregation subtree in the system. Then we describe how our system uses a self-tuning algorithm to address the policy question of distributing an AI budget across subtrees to minimize system load.

3.1.1 Mechanism

To enforce AI, each aggregation subtree T for an attribute has an error budget δ_T which defines the maximum inaccuracy of any result the subtree will report to its parent for that attribute. The root of each subtree divides this error budget among itself δ_{self} and its children δ_c, and the children recursively do the same. Here we present the AI mechanism for the SUM aggregate; other standard aggregation functions (e.g., MAX, MIN, AVG) are described in the appendix.

This arrangement reduces system load by filtering small updates that fall within the range of values "cached" by a subtree's parent. In particular, after a node A with error budget δ_T reports a range $[V_{min}, V_{max}]$ for an attribute value to its parent (where $V_{max} = V_{min} + \delta_T$), if the node A receives an update from a child, the node A can skip updating its parent as long as it can ensure that the true value of the attribute for the subtree lies between V_{min} and V_{max}, i.e., if

$$\begin{array}{rcl} V_{min} & \leq & \sum_{c \in children} V^c_{min} \\ V_{max} & \geq & \sum_{c \in children} V^c_{max} \end{array} \qquad (1)$$

where V^c_{min} and V^c_{max} denote the most recent update received from child c.

Notice the trade-off in splitting δ_T between δ_{self} and δ_c. Large values of δ_c allow children to filter updates before they reach a node. Conversely, by setting $\delta_{self} > 0$, a node can set $V_{min} < \sum V^c_{min}$, set $V_{max} > \sum V^c_{max}$, or both to avoid further propagating some updates it receives from its children.

PRISM maintains per-attribute δ values so that different attributes with different error requirements and different update patterns can use different δ budgets in different subtrees. PRISM implements this mechanism by defining a per-attribute-type *distribution function* that is analogous to the per-attribute-type aggregation function. Just as an attribute type's aggregation function specifies how aggregate values are aggregated from children, an attribute type's distribution value specifies how δ budgets are distributed among children and δ_{self}.

3.1.2 Policies

Given the above mechanisms, to guarantee that the total aggregation error does not exceed the root error budget δ_{root} for an attribute, we just need to ensure that the following two conditions hold at the root node of every subtree T.

$$\begin{array}{rcl} \delta_T & \geq & \delta_{self} + \sum_{c \in children} \delta_c \\ V_{max} & \leq & V_{min} + \delta_T \end{array} \qquad (2)$$

Given these constraints, we still have plenty of freedom to (i) set δ_{root} to an appropriate value for each attribute, (ii) compute V_{min} and V_{max} when updating a parent, and (iii) split δ into δ_{self} and δ_c. Below we present policies that exploit such freedom to optimize the precision v. performance trade-off.

Setting δ_{root}. Note that the aggregation queries can set the root error budget δ_{root} to any non-negative value. For some applications, an absolute constant value may be known a priori (e.g., count the number of connections per second ± 10 at port 1433.) For other applications, it may be appropriate to set the tolerance based on measured behavior of the aggregate in question (e.g., set δ_{root} for an attribute to be at most 10% of the maximum value observed) or the measurements of a set of aggregates (e.g., in our heavy hitter application, set δ_{root} for each flow to be at most 1% of the bandwidth of the largest flow measured in the system). Our algorithm supports all of these approaches by allowing new absolute δ_{root} values to be introduced at any time, and we have prototyped systems that use each of these three policies.

Computing [V_{min}, V_{max}]. When either $\sum_c V^c_{min}$ or $\sum_c V^c_{max}$ goes outside of the last [V_{min}, V_{max}] that was reported to the parent, a node needs to report a new range to its parent. Given a δ_{self} budget at an internal node, we have some flexibility on how to center the $[V_{min}, V_{max}]$ range. Our approach is to adopt a per-aggregation-function range policy that reports $V_{min} = (\sum_c V^c_{min}) - bias * \delta_{self}$ and $V_{max} = (\sum_c V^c_{max}) + (1 - bias) * \delta_{self}$ to the parent. The *bias* parameter can be set as follows: Set

- $bias \approx 0.5$ if inputs expected to be roughly stationary
- $bias \approx 0$ if inputs expected to be generally increasing
- $bias \approx 1$ if inputs expected to be generally decreasing

For example, suppose a node with total δ_T of 10 and δ_{self} of 3 has two children reporting ($[V^c_{min}, V^c_{max}]$) of [1, 2] and [2, 8], respectively, and reports [0, 10] to its parent. Then, the first child reports a new range [10, 11], so the node must report to its parent a range that includes [12, 19]. If $bias = 0.5$, then report to parent [10.5, 20.5] to filter out small deviation around the current position. Conversely, if $bias = 0$, report [12, 22] to filter out the maximal number of updates of increasing values.

Self-tuning error budgets. The final policy question is how to divide a given error budget δ_{root} across the nodes in an aggregation tree.

A simple approach is to have a static policy that divides the error budget *uniformly* among all the children. For example, a node with budget δ_T could set $\delta_{self} = 0.1\delta$ and then divide the remaining $0.9\delta_T$ evenly among its children. Although this approach is simple, it is likely to be inefficient because different aggregation subtrees may experience different loads.

Algorithm. To make cost/accuracy tradeoffs *self-tuning*, PRISM provides an adaptive algorithm by which nodes adjust to changing error budget δ_T and adapt the balance between δ_{self} and δ_c for each child c. The high-level idea is simple: increase δ for nodes with high load and low δ and decrease δ for nodes with low load and high δ. Unfortunately, a naive rebalancing algorithm could easily spend more network messages redistributing δs than it saves by filtering updates. This is a particular concern for applications like distributed heavy hitter that monitors a large number of attributes, only a few of which are active enough to be worth optimizing. To address this challenge PRISM uses a two-step algorithm:

1. *Estimate optimal distribution of δ_T among δ_{self} and δ_c.*

Each node tracks the number of messages sent to its parent per time unit (M_{self}) and the aggregate number of updates per time unit reported by each child c's subtree (M_c). Note that M_c reports are accumulated by a child until they can be piggy-backed on an update message to its parent. Given this information each node n estimates the optimal values δ_v^{opt} that minimizes the total system load $\sum_v M_v^{opt}$, where M_v^{opt} is an estimate of the load generated by node v under optimal error budget δ_v^{opt}. In particular, for any $v \in \{self\} \cup child(n)$ we estimate

$$\delta_v^{opt} = \delta_T * \frac{\sqrt{M_v * \delta_v}}{\sum_{v \in \{self\} \cup child(n)} \sqrt{M_v * \delta_v}}. \quad (3)$$

which is optimal assuming that load is inversely proportional to error budget and which seems a reasonable heuristic for predicting the impact of small changes.

2. *Redistribute deltas iff the expected benefit exceeds the redistribution overhead.*

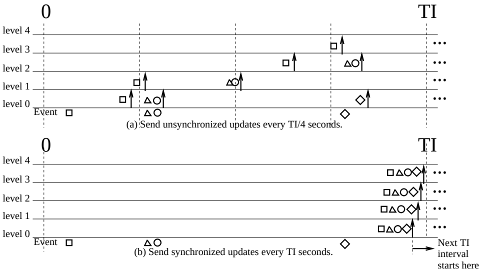
At any time, a node n computes a *charge* metric for each child subtree c, which estimates the number of extra messages sent by c due to sub-optimal δ. $Charge_c = (T_{curr} - T_{adjust}) * (M_c - M_c^{opt})$, where T_{adjust} is the last time δ was adjusted at n. Notice that a subtree's charge will be large if (a) there is a large load imbalance (e.g., $M_c - M_c^{opt}$ is large) or (b) there is a stable, long-lasting imbalance (e.g., $T_{curr} - T_{adjust}$ is large.)

We only send messages to redistribute deltas if doing so is likely to save at least k messages (i.e., if $charge_c > k$). To ensure the invariant that $\delta_T \leq \delta_{self} + \sum_c \delta_c$, we make this adjustment in two steps. First, we loan some of the δ_{self} budget to the node c that has accumulated the largest charge by incrementing c's budget by min(0.1δ_c, max(0.1δ_{self}, δ_{self} - δ_{self}^{opt})). Second, we replenish δ_{self} from the child whose δ_c is the farthest above $delta_c^{opt}$ by ordering c to reduce δ_c by min(0.1 δ_c, δ_c - $delta_c^{opt}$).

A node responds to a request from its parent to update δ_T using a similar approach.

3.2 Temporal Imprecision

Temporal imprecision provides a real-time bound on the delay between when an update occurs at a leaf node and when it is reflected in the aggregated result reported by the root. A temporal imprecision of TI seconds guarantees that every event that occurred TI or more seconds ago is reflected in the reported result; events younger than TI may or may not be reflected. [32].



(a) Send unsynchronized updates every TI/4 seconds.

(b) Send synchronized updates every TI seconds.

Fig. 2: For a given TI bound, pipelined delays with synchronized clocks (b) allows nodes to send less frequently than unpipelined delays without synchronized clocks (a).

Temporal imprecision benefits monitoring applications in two ways. First, it accounts for inherent network and processing delays in the system; given a worst case per-hop cost hop_{max} even immediate propagation provides a temporal guarantee no better than $\ell * hop_{max}$ where ℓ is the maximum number of hops from any leaf to the root of the tree. Note that although Internet round trip times have a very long tail [2, 8, 24], the network imprecision metric allows us to *assume* a relatively low hop_{max} (e.g., 10 seconds) because if network and processing times increase beyond this bound, then the network imprecision metric reflects the unexpected delay.

Second, explicitly exposing TI provides an opportunity to combine multiple updates to improve scalability by reducing processing and network load. If the TI guarantee for an attribute exceeds the minimum system latency i.e., $TI > \ell * hop_{max}$ then a node in tree can use the "extra" time to try to accumulate multiple updates from the node's children before calculating and sending a single update to the parent.

Below we present an optimized mechanism for implementing temporal imprecision using *pipelined delays* based on synchronized clocks. PRISM also provides a fall-back alternative for unsynchronized clocks [18].

Pipelined delays. We maximize the opportunity for batching updates by *pipelining* the available slack delay across levels of the aggregation hierarchy.

Suppose clocks were perfectly synchronized and suppose that message transmission and processing were instantaneous. Then, as Figure 2(a) illustrates, one option to enforce a bound of TI for an ℓ-level tree (ℓ = 4 in Figure 2) would be for every node to send an update to its parents every TI/ℓ seconds. Alternatively, as Figure 2(b) illustrates, each leaf node could send a batch update (combining all its updates in the current TI interval) at time $TI - \ell * \epsilon$, all level-1 nodes at time $TI - (\ell - 1) * \epsilon$, and so on for some small ϵ. Note that the next TI interval starts after $TI - \ell * \epsilon$ time in the current interval effectively leading to a batching interval of $TI - \ell * \epsilon$ at every level. Thus, by synchronizing update transmission times across levels, we still meet the TI guarantee but increase the batching interval from TI/ℓ to $TI - \ell * \epsilon$.

Of course, a real implementation must account for clock skew, processing delays, and network delays to ensure that level i holds opens a sufficient window of time for level $i-1$'s updates to arrive and be processed. For the leaves, we specify a send interval I and an arbitrary reference time Z such that for the jth interval at time $Z + j * I$ a leaf node sends an update if and only if the value has changed since $Z + (j-1) * I$. We calculate I as follows: (1) we synchronize the clocks on different nodes such that the maximum skew between any two nodes is $skew_{max}$.[1] (2) We define a "stagger" parameter S that bounds the delay for updates to traverse levels, i.e., $S = hop_{max} + 2 * skew_{max}$. Finally, (3) we set $I = TI - \ell * S$. Note that the smallest TI that can be provided is $TI_{min} = \ell * S$. Also note that for load balancing, different attributes can choose different Z base times.

For internal nodes, we pipeline each level's timeouts to occur just before its parent's. Specifically, at time $Z + j * I + i * S$ an aggregation node at level i sends an updated aggregate value to its parent if and only if the value of any of its inputs has changed since time $Z + (j-1) * I + i * S$. This approach ensures the following property: an event at a leaf node at local time X is reflected at root no later than time $X + TI$ according to the local time at the leaf node.

4 Network Imprecision

Network imprecision characterizes the uncertainty introduced by node crashes, slow network paths, unreachable nodes, and DHT topology reconfigurations. In particular, if a a subtree is silent over an interval, a aggregation system must distinguish two cases: (1) the subtree has sent no updates because the inputs have not significantly changed versus (2) the inputs have significantly changed but the subtree is unable to transmit its report. This problem is fundamental, and it is an extension of the CAP impossibility result [13, 31], but it is made worse by the *amplification effect* of hierarchical aggregation: 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. If these issues are not addressed by an aggregation system, the results a monitoring system reports may be *arbitrarily incorrect.*

The key idea of NI is that because no system can guarantee to always provide the "right" answer, it instead must report the extent to which a calculation could have been disrupted by network and node problems.This information allows applications to filter out or take action to correct measurements with unacceptable uncertainty. To that end, NI is composed of three metrics, N_{all}, $N_{reachable}$, and N_{dup}.

- N_{all} is an estimate of the number of nodes that are members of the system.
- $N_{reachable}$ is a lower bound on the number of nodes for which input propagation is guaranteed to meet an attribute's TI bound.
- N_{dup} provides an upper bound on the number of nodes whose contribution to an attribute may be doubly-counted. Double-counting can occur when reconfiguration of an aggregation tree's topology causes a leaf node or virtual internal node to fail-over to a new parent while its old parent retains the node's inputs as soft state until a timeout.

Conditioned Consistency. These three metrics condition the arithmetic and temporal consistency guarantees. In particular, reading an attribute's value from the system returns a tuple $[V_{min}, V_{max}, TI, [N_{all}, N_{reachable}, N_{dup}]]$ that means "The system estimates the value to be between V_{min} and V_{max}. This estimate may omit some inputs that occurred in the last TI seconds and it may also omit some inputs from $N_{all} - N_{reachable}$ of the N_{all} nodes in the system. This estimate may double count inputs from at most N_{dup} nodes."

Users and applications must evaluate the significance of disruptions that cause $N_{reachable} < N_{all}$ or $N_{dup} > 0$ in the context of their requirements. For some applications, an aggregate result may be unusable if it omits or duplicates any inputs. Conversely, other applications may be content with best-effort results and may ignore NI completely. Other applications will take a middle ground and be structured to tolerate modest amounts of NI.

In Section 5.3, we explore one general and effective strategy: using redundant trees in the DHT to compute an attribute and then using NI to identify the highest-quality result. Other available approaches include constructing aggregates that tolerate common forms of NI (e.g., the MAX aggregation function is insensitive to $N_{dup} > 0$), taking corrective action during periods of high NI (e.g., actively probing unresponsive machines, triggering on-demand reaggregation [40], considering the recent history of aggregate values, or delaying taking action until more conclusive information is gathered), or flagging results as GOOD/MARGINAL/BAD depending on the observed NI.

Challenges. Although monitoring connectivity to nodes to compute the NI metrics appears straightforward—the NI metrics are all conceptually

[1] Algorithms in the literature can achieve clock synchronization among nodes to within one millisecond [37].

aggregates across the state of the system—in practice two challenges arise. First, the system must cope with reconfiguration of dynamically-constructed aggregation trees. 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.

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} in order to account for dynamically changing aggregation topologies. Finally, in Section 4.3 we describe how to scale the approach to work with the 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, and it 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 this aggregate remains cached as soft state by its ancestors for a long timeout $T_{declareDead}$.

$N_{reachable}$ 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 in which no hop takes longer than hop_{max}. 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 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 a connection to a subtree is no longer a good path. Each internal node pings its child once every p time units and maintains $sendPingReplied_c$, the time it sent the last ping for which it has received a reply from c. A child sends its ping reply only after sending any messages backlogged in its outbound message queue. Note that p should be smaller than hop_{max}; we use $p = 10$ seconds by default (hop_{max} = 30 seconds). If $sendPingReplied_c + hop_{max} < curr_{time}$ then a node declares child c unreachable: the node removes c's subtree contribution from the $N_{reachable}$ aggregate and immediately sends the new value up towards the root of the $N_{reachable}$ aggregation tree. Notice that for simplicity this approach is conservative—it declares a child "unreachable" if the round trip time (rather than the one way time) exceeds hop_{max}.

Nreachable v. TI $N_{reachable}$ characterizes the current topology of the aggregation tree, but past connectivity disruptions could affect attributes with large TI. In particular, because our TI algorithm defines a small window of time during which a node must propagate updates to its parents, then any attribute's subtree that was unreachable over the last TI_{attr} could have been unlucky and missed its window even though the subtrees nodes are currently all counted as reachable. We must either (a) modify the protocol to ensure that such a subtree's updates are reflected in the aggregate so that the promised TI bound is met or (b) we must ensure that $N_{reachable}$ counts such subtrees as unreachable because they may have violated their TI bound.

We take the former approach to avoid having to calculate a multitude of $N_{reachable}$ values for different TI bounds. In particular, when a node receives updates from a child marked unreachable, it knows those updates may be late and may have missed their window for TI propagation. 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 TI delays are temporarily ignored 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 compute new values to pass up using local information. This information is soft state—a parent discards it if a client 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" [34]), we use long timeouts to avoid spurious garbage collection (e.g., we use $T_{declareDead} \approx 10$ minutes in our prototype.)

Notice that when a subtree chooses a new parent, then that subtree's inputs may still be stored by a former parent and thus be counted multiple times in the aggregate. N_{dup} bounds the number of leaf inputs that might be included multiple times in an aggregate calculation.

The basic aggregation function for N_{dup} is simple. We keep a count k of the number of leaves in each subtree using the obvious aggregation function. Then, 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 sufficient time has elapsed that its old parent has safely removed its soft state, it updates

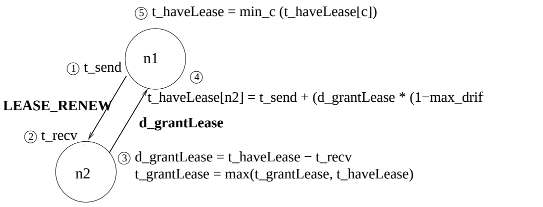

⑤ t_haveLease = min_c (t_haveLease[c])
n1
① t_send
④
LEASE_RENEW
t_haveLease[n2] = t_send + (d_grantLease * (1−max_drif
d_grantLease
② t_recv
n2
③ d_grantLease = t_haveLease − t_recv
t_grantLease = max(t_grantLease, t_haveLease)

Fig. 3: Protocol for a parent to renew a lease on the right to hold as soft state a child's contribution to an aggregate.

its input of N_{dup} to 0.

Lease aggregation. For correctness, our implementation uses a *lease aggregation* algorithm that extends the concept of leases [14] to hierarchical aggregation.

Figure 3 illustrates the protocol used when a node n_1 updates a lease on the right to cache 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 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 a child 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 the node.

To avoid spurious lease expirations, each node renews leases from its descendants once every $renew$ seconds and leaf nodes grant leases of length $T_{declareDead}$ with $renew << T_{declareDead}$ (e.g., $renew$ = 30 seconds and $T_{declareDead}$ = 10 minutes in our prototype).

Early expiration. 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 4(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 re-adding large subtrees of inputs. Furthermore, note that the example in Figure 4 is the common case: in a randomly constructed tree, the vast majority of nodes (and failures) are near the leaves. Failing to address this problem would transform the common-case of leaf failures into significant disruptions near the root and bring into play the amplification effect.

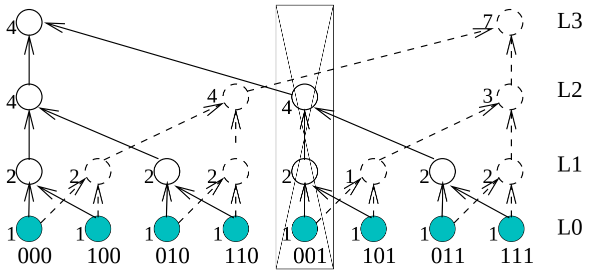


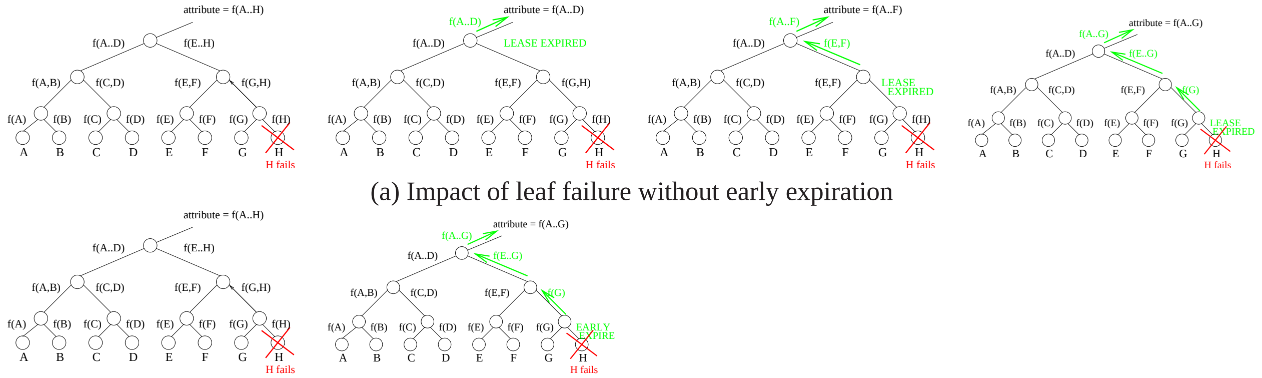
Fig. 5: The failure of a physical node has different effects on different aggregations depending on which virtual nodes are mapped to the failed physical node. The numbers next to virtual nodes show the value of $N_{reachable}$ for each subtree after the failure of physical node 001, which acts as a leaf for one tree but as a level-2 subtree root for another.

Early expiration avoids this unwarranted disruption as Figure 4(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 monitoring to large numbers of nodes and attributes, PRISM constructs a forest of trees using an underyling DHT and then uses different aggregation trees for different attributes. As Figure 5 illustrates, a failure affects different trees differently so we need to calculate 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 active probing every p seconds along each edge of each tree's graph.

As a result of these factors, the straightforward algorithm for maintaining NI metrics separately for each tree is not tenable: n degree-d trees each with $\theta(\frac{n-1/d}{1-1/d})$ nodes have $\theta(dn^2)$ edges that must be monitored; such monitoring would require $\theta(dn^2)$ messages per node every p seconds (p = 10 in our system). To put this in perspective, consider a n = 512-node system with d = 8-



(a) Impact of leaf failure without early expiration

(b) Impact of leaf failure with early expiration

Fig. 4: Recalculation of aggregate function across values A, B, ..., H after the node with input H fails (a) without and (b) with early expiration.

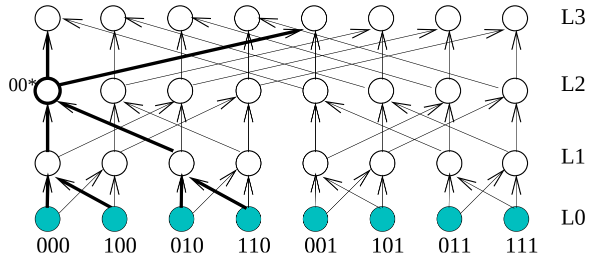


Fig. 6: Plaxton tree topology is an approximate butterfly network. The bold connections illustrate how a virtual node 00* uses the dual tree prefix aggregation abstraction to aggregate values from a tree below it and distribute the results up a tree above it.

ary trees (i.e., a DHT with 3-bit correction per hop). The straightforward algorithm then has each node sending over roughly 400 pings per second. As the system grows, the situation deteriorates rapidly—a 4096-node system requires each node to send roughly 3200 pings per second.

Our solution reduces active monitoring work to $\theta(d \log n)$ pings per node per p seconds. The 512-node system in the example will require each node to send about 3 pings per second; the 4096-node system will require each node to send about 4 pings per second.

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 [27] to re-use common sub-calculations across different aggregation trees using a novel *dual tree prefix aggregation* abstraction.

In particular, we note that as Figure 6 illustrates, the Plaxton tree algorithm 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 in that tree 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 an aggregation 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 6 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 the $N_{reachable}$ count and N_{dup} lease, the values propagated up are aggregates on the subtree (the number of reachable nodes and the minimum lease duration granted by the subtree), so the same value can be propagated by a node to all of its parents.

For example in Figure 6, 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 descendents 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 three bits per hop and d=8, so each subtree is common to 8 parents.

5 Experimental Evaluation

To evaluate PRISM, we perform experiments on two types of networks: (1) several LAN clusters (a 50-node departmental Condor cluster and 50 to 85 Emulab [39] nodes) and (2) 94 nodes on the PlanetLab distributed testbed [26]. Our prototype has been developed using SDIMS [40] on top of FreePastry [29].

Our experiments characterize the performance and scalability of the AI, TI, and NI metrics for PrMon and distributed heavy hitters (DHH) applications. First, we

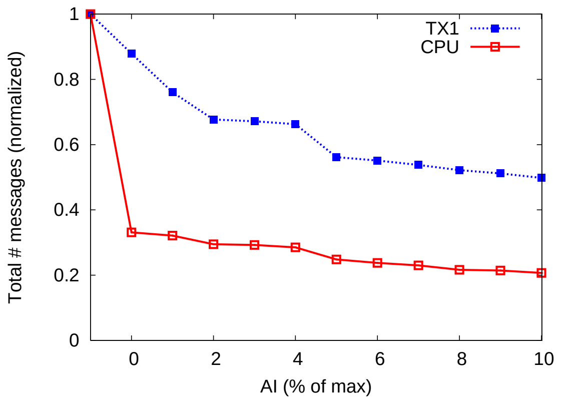


Fig. 7: Load vs. AI for TX1 and CPU attributes with no TI filtering

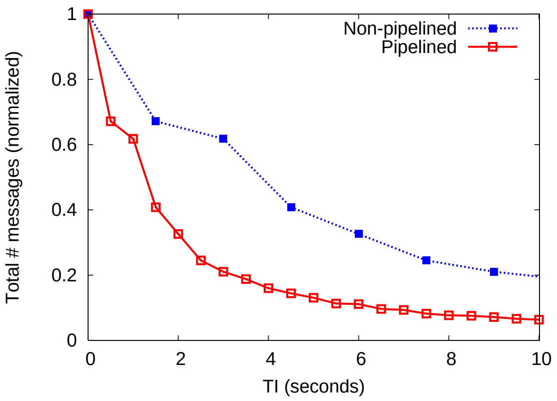


Fig. 8: Load vs. TI for a single attribute with no AI filtering

use CoMon [6] data collected from PlanetLab [26] and netflow traces from Abilene [1] to quantify the reduction in monitoring overheads due to AI and TI. Second, we analyze the deviation in the PRISM's reported values with respect to both the ground truth based on sensor readings and the guarantees defined by AI and TI. Finally, we investigate the consistency/availability tradeoffs that NI exposes. In summary, our experimental results show that PRISM is an effective substrate for scalable monitoring: introducing small amounts of AI and TI significantly reduces monitoring load, and the NI metrics both successfully characterize system state and reduce measurement inaccuracy.

5.1 Load vs. Imprecision

In this subsection we quantify the reduction in monitoring load due to AI and TI for both the PrMon and DHH applications. Further, we characterize the reduction in monitoring load due to AI and TI for different sensor data distributions by running large-scale simulations on synthetic datasets.

5.1.1 PrMon

We begin by comparing the monitoring cost of PrMon distributed monitoring service to the centralized CoMon service, which uses a fixed TI of 5 minutes and which does not exploit AI. We gather CoTop [6] data from 200 PlanetLab nodes at 1-second intervals for 1 hour on 25 September 2006. The CoTop data provides the per-slice resource usage (e.g., CPU, MEM, TX1) for all slices running on a given PlanetLab node. Using these logs as sensor input, we run PRISM on 200 servers mapped to 50 Emulab machines each having a 3GHz CPU and 2GB RAM.

Figure 7 shows the AI precision-performance results for the PrMon application for two attributes (the total TX1 and CPU usage of slice princeton_codeen across 200 PlanetLab nodes). The TX1 attribute denotes the total number of bytes transmitted by a slice in the last minute. The x-axis shows the global AI budget, and the y-axis shows the total message load normalized with respect to AI of -1 (no AI caching) and TI = TI_{min} = 50ms. Each data point represents the total number of messages sent during the 1-hour run. From the figure, we observe that for CPU, the load falls by 68% when AI changes from -1 (no caching) to 0 and a 10% AI further provides almost a 40% reduction in load compared to AI=0. The load reduction from AI=-1 to AI=0 comes from culling new updates that exactly match the previous report. However, if the CPU value changes, it generally deviates by a large amount, resulting in limited gains achieved by 10% AI. For the TX1 attribute, the sensor sends an update every 60 seconds. In this case, changing AI from -1 to 0 provides roughly a 12% reduction in load whereas 10% AI reduces the load by 50%.

Figure 8 shows the corresponding TI precision-performance results with no AI filtering. The initial TI value of TI_{min} (50 ms) corresponds to immediate propagation of messages along the aggregation tree. From the graph, we observe that the reduction in system load is 80% and over an order of magnitude for non-pipelined and pipelined 10 second TI delays respectively compared to TI of TI_{min}.

Figure 9 shows the combined effect of AI and TI in reducing monitoring load for the CPU attribute for the princeton_codeen slice. We use TI of 10 seconds, 30 seconds, 1 minute, and 5 minutes, and for each of these TI values, we run the experiment for AI values of -1, 0, 10%, and 20%. We observe that the load falls by 70% from AI of -1 to AI of 10% for a given TI. Further, for a fixed AI, the monitoring load shows a curve following 1/TI as in Figure 8. For this attribute, giving an AI of 10% or 20% only provides additional load reduction of 10% and 16% respectively due to low temporal locality.

Next Figure 10 shows the combined effect of AI and TI in reducing monitoring load for all the nine attributes (TX1, TX15, RX1, RX15, #PR, PMEMMB, VMEMMB, CPU%, and MEM%) emitted by the CoTop

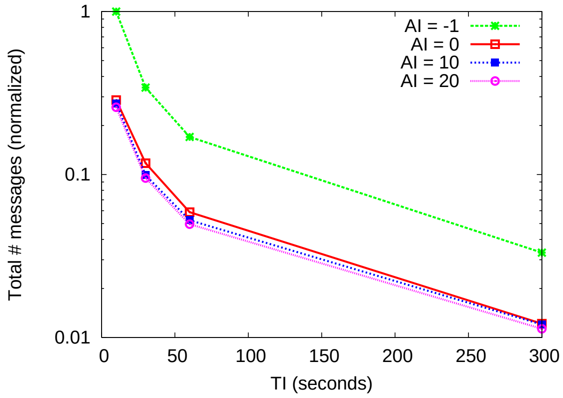


Fig. 9: Load vs. AI and TI for CPU attribute

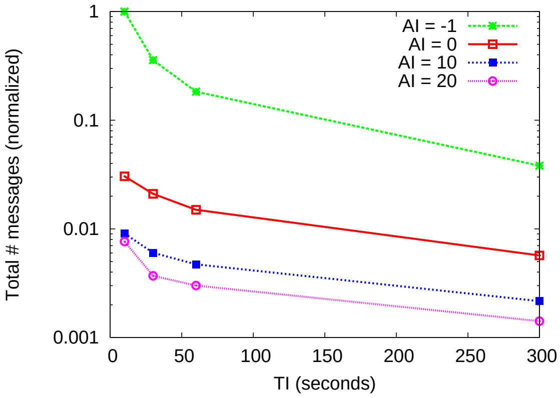


Fig. 10: Load vs. AI and TI for all attributes

sensor for all the running PlanetLab slices in our trace data. We observe that for AI of -1, there is more than one order of magnitude load reduction for TI of 5 minutes compared to 10 seconds. Likewise, for a fixed TI of 10 seconds, AI of 20% reduces load by two orders of magnitude compared to AI = -1. By combining AI of 20% and TI of 30 seconds, we get both an order of magnitude load reduction and an order of magnitude reduction in the time lag between updates compared to CoMon's AI of -1 and TI of 5 minutes. Alternatively, for approximately the same bandwidth cost as CoMon with TI of 5 minutes and AI of -1 for 200 nodes, PRISM provides highly time-responsive and accurate monitoring with TI of 10 seconds and AI of 0.

5.1.2 Detecting Heavy Hitters

For our heavy hitter case study, we use multiple netflow traces obtained from Abilene [1] Internet2 backbone network. The data was collected for 1 hour on April 4, 2006; each backbone router logged per-flow data every 5 minutes, and we split this trace into 200 buckets based on the hash of source IP. Our monitoring system executes a Top-10 query on this dataset for tracking the top 10 flows (destination IP as key) in terms of bytes received over a 30 second moving window shifted every 5 seconds.

Figure 11 shows the precision-performance results for the top-10 heavy hitter query for 50 nodes on the departmental Condor testbed. The x-axis shows the AI budget and the y-axis shows the total monitoring load per unit time normalized relative to the load for AI = 0. The AI budget is varied from 1% to 10% of the top flow's global traffic volume. From the graph, we observe that the knee of the graph at 10% AI provides over an order of magnitude reduction in monitoring load. A large fraction of the reduction comes from completely eliminating aggregation for "mouse" flows whose total bandwidth is less than the imprecision budget at the leaves.

Figure 12 shows the corresponding results for the pipelined and non-pipelined TI delays. We find that using pipelined delays, a 10 seconds TI achieves an order of magnitude reduction in monitoring load. Increasing the TI beyond 10 seconds yields additional, albeit smaller, reductions. For non-pipelined delays, TI of 25 seconds yields an order of magnitude load reduction.

5.1.3 Generalized Model: Simulation study

To generalize the trade-off between AI and monitoring cost, we evaluate the conditions under which AI is effective i.e., the distribution of the data values reported by the sensors. We first investigate via simulation a large-scale aggregation network with 7776 physical nodes organized as a 5 level aggregation tree with uniform degree 6. For each leaf node, we model the the data values of incoming traffic using two distributions: a Gaussian distribution and a uniform distribution. Our aim is to the evaluate the effect on load due to the noise in the data values given a fixed AI budget.

Figure 13 shows the corresponding results using our simulator over these two data distributions. The x-axis denotes the ratio of total noise induced by all the leaf sensors to the total AI budget. We observe that when noise is small compared to the AI budget, we filter almost all updates and load can be reduced by an order of magnitude. But, as expected, when noise is large compared to the error budget, the load asymptotically approaches the load with AI = 0. The uniform distribution allows almost perfect culling of updates for small amounts of noise whereas for the Gaussian distribution, there is a small yet a finite probability for data values to deviate arbitrarily from their previously reported range.

In summary, our evaluation shows that small AI and TI budgets can provide large bandwidth savings to enable scalable monitoring.

5.2 Promised vs. Realized Accuracy

In this subsection we aim to answer the following question: do PRISM's reported values reflect reality? We quantify the difference between PRISM's reports and the "ground truth" based on the instantaneous sensor readings. To evaluate this deviation, we compare PRISM's

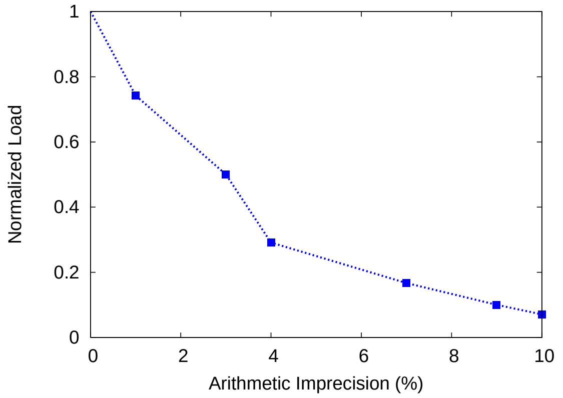


Fig. 11: Normalized load vs. AI for the top-10 query on Abilene traces.

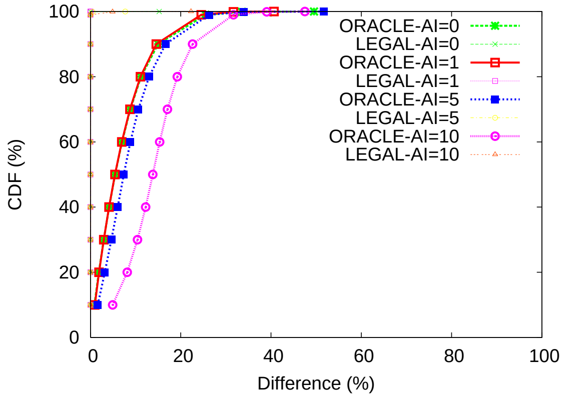


Fig. 14: Cumulative Distribution function (CDF) for difference between PRISM's reported values wrt. (a) oracle's reports and (b) PRISM's legal guarantees for fixed TI of 1 second.

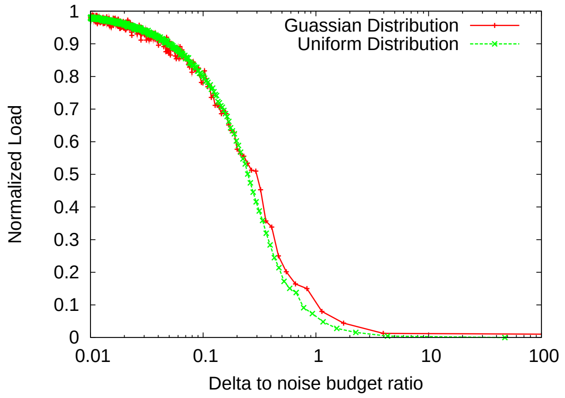


Fig. 13: Normalized Load vs. noise of synthetic workload for a fixed AI budget. If noise $<$ AI, a majority of updates get filtered.

reported results with (1) an oracle service that reports true aggregate values based on sensor readings at any time instant and (2) the legal guarantees promised by PRISM's AI and TI metrics.

Figure 14 and 15 show the CDF of deviation between PRISM's reported values for PrMon's "CPU" attribute compared to both the "oracle" instantaneous values and

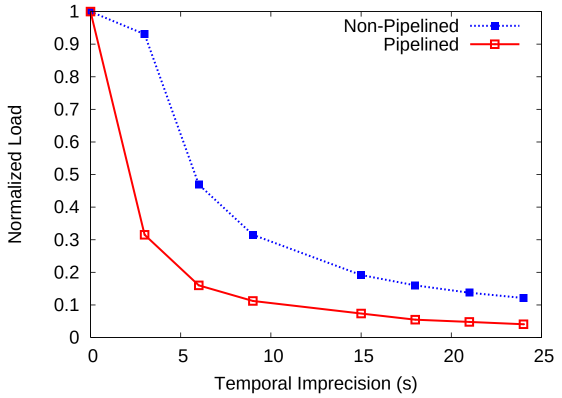


Fig. 12: Normalized load vs. TI for the top-10 query on Abilene traces.

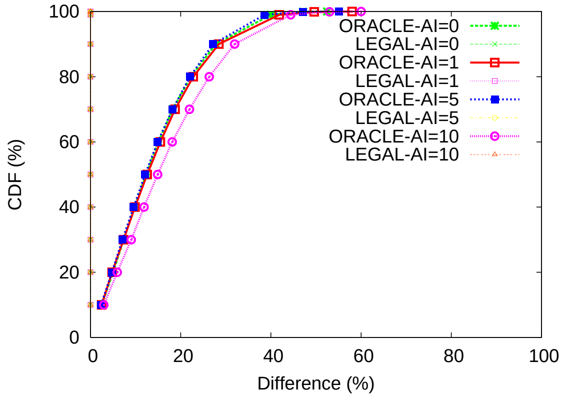


Fig. 15: Cumulative Distribution function (CDF) for difference between PRISM's reported values wrt. (a) oracle's reports and (b) PRISM's legal guarantees for fixed TI of 10 seconds.

the legal guarantees defined by AI and TI. In Figure 14, we fix TI to 1 second and then report the CDF of difference in the attribute's reported values for different values of AI. We make two important observations here: (1) PRISM's reported values lie within the envelope defined by AI and TI for essentially all reports and (2) for 5% AI and 1 second TI, more than 90% of reports have difference less than 15% from the oracle. As illustrated in Figure 14, increasing the TI to 10 seconds results in larger deviation between PRISM's reported results and the oracle. For 5% AI and 10s TI, more than 90% reports differ by less than 27% from the oracle. The relatively large errors relative to AI are due to the low temporal locality of the CPU attribute: small TI adds significant additional variation compared to the oracle. But, the values remain within the legal guarantees defined by the combined AI and TI limits.

5.3 NI: Coping with Disruption

Finally, we analyze the effectiveness of our NI metrics in accurately reflecting network state and filtering inaccurate reports.

We first show how NI metrics reflect network state for

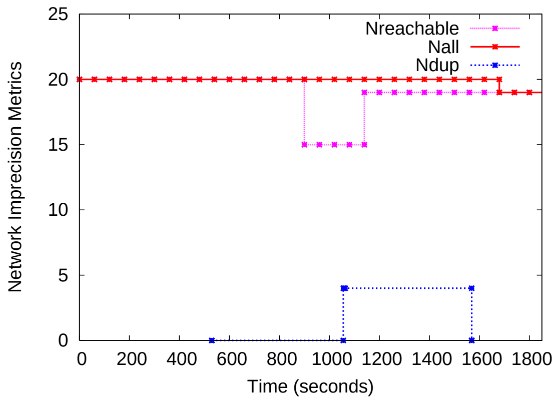


Fig. 16: NI metrics under induced system churn – single node failure at 815 seconds into the experimental run.

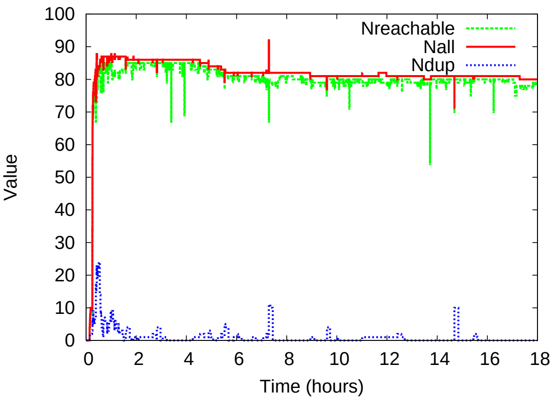


Fig. 17: NI metrics reflecting PlanetLab state (85 nodes).

a small scale controlled experiment. In Figure 16, we run a 20 node experiment on the Condor cluster where we kill a single node at 815 seconds 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. The N_{all} value remain stable from 20 until about 1600 seconds to reflect the long $T_{declareDead}$ timeout before the system declares unreachable nodes to be dead. Correspondingly, the N_{dup} value goes from 0 to 4 at about 1060 seconds when the disconnected subtree joins a new parent and starts reporting its N_{dup} value to that parent. Finally, the N_{dup} value falls back to 0 about $T_{declareDead}$ time units ($T_{declareDead}$ = 10 minutes) after the dead event and both N_{all} and $N_{reachable}$ stabilize to 19 (nodes) denoting that the system is back to a stable state.

Figure 17 shows how NI reflects network state for a 85-node PlanetLab experiment for a 18-hour run starting 4 October 2006. 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 to 80.

Next we quantify the risks of reporting global aggregate results without incorporating NI. We run a 1 hour experiment on 94 PlanetLab nodes for an attribute with AI = 0 and TI = 10 seconds. Figure 18 shows the CDF of reported answers showing the deviation in reports with respect to an oracle. 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 and when NI $<$ 90%, 80% answers can deviate by as high as 65% from the true value. Note that for monitoring systems that ignore NI (no filtering line), 90% of their reports can differ by 80% from the truth.

In Figure 19 we explore the effectiveness of a general strategy to achieve high consistency in reported aggregate values during periods of churn. We use K redundant trees in the DHT to compute an attribute and then use NI to identify the highest-quality result. Figure 19 shows the CDF of results with respect to the deviation from oracle as we vary K from 1 to 4. We observe that 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.

Filtering answers during periods of high churn exposes a fundamental consistency versus availability tradeoff [13]. Figure 20 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 behind the approach is that since the vast majority of nodes 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 the appendix.

6 Related Work

The three imprecision metrics in our work are inspired by and relate to a number of research traditions in the distributed systems community.

The AI and TI metrics are related to several research efforts allow applications to trade precision for commu-

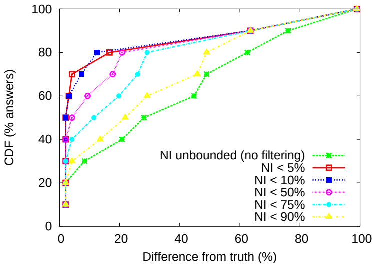


Fig. 18: Cumulative Distribution function (CDF) for reported answers filtered for different NI thresholds and K = 1.

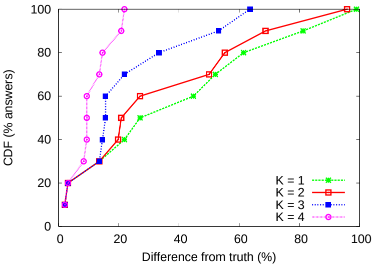


Fig. 19: Cumulative Distribution function (CDF) of NI values for different K.

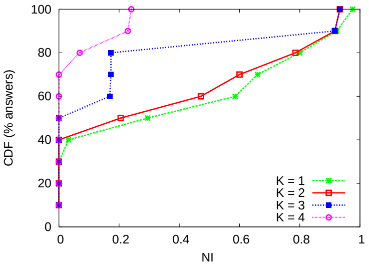


Fig. 20: Cumulative Distribution function (CDF) of NI values for K duplicate keys.

nication overhead. Olston et al. [3, 22] propose adaptive filters at the data sources that compute approximate answers for continuous queries. Their work, however, focuses on single-level communication topologies. In a hierarchical communication setting, Manjhi et al. [21] consider the problem of finding frequent items in database streams; they focus on determining an optimal but *static* distribution of slack to the internal and leaf nodes of the tree. TAG [20], an aggregation service for sensor networks, employs a similar approach as PRISM for bounding TI when nodes are approximately synchronized.

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. Similarly, file systems providing cache consistency often provide leases on individual objects [14] or volume leases on groups of objects [43]

Consistency for aggregation differs in two fundamental ways. First, aggregation systems are large-scale with many concurrent writers which implies that it is not feasible to resolve CAP dilemma [13] 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. Second, in hierarchical aggregation that accumulates inputs from many sensors, amplification effect of failures can make results substantially deviate from the real values.

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 conditioned consistency is similar in spirit to the notion of failure detectors [4] for fault-tolerant distributed systems. Also, in quorum systems, Pierce and Alvisi's pseudo-regular and pseudo-atomic semantics provide regular and atomic semantics on all operations, but they allow operations to *abort* if concurrency or network failures would prevent such a guarantee [25]. Kostoulas et al. [19] point out the impossibility of group size estimation in a dynamic group and propose an active gossip-based scheme and a passive approach based on interval densities when nodes are hashed onto a given real interval. 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, designed for groups on the scale of tens of nodes, monitors the nodes and system and exposes such attestation graph to the applications.

Traditionally, DHT-based aggregation is event-driven and best-effort, i.e., each update event triggers re-aggregation for affected portions of the aggregation tree. Further, systems often only provide eventual consistency guarantees on its data [36, 40], i.e., updates by a live node will eventually be visible to probes by connected nodes.

There are ongoing efforts similar to ours in the P2P and databases community to build global monitoring services. PIER is a DHT-based relational query engine [16] targeted at querying real-time data from many vantage-points on the Internet. Sophia [38] is a distributed monitoring system designed with a declarative logic programming model. A recent study [17] has proposed support of aggregate triggers in monitoring systems in which individual nodes can independently detect and react to changes in the global system-wide behavior. PRISM may enhance such efforts by providing a scalable way to track top-k and other significant events.

7 Conclusions

Without precision guarantees, large scale network monitoring systems may be too expensive to implement (because too many events flow through the system) or too dangerous to use (because data output by such systems may be arbitrarily wrong.) PRISM provides arithmetic imprecision to bound numerical accuracy, temporal imprecision to bound staleness, and network imprecision to expose cases when first two bounds can not be trusted.

References

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

[2] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Re-

silient overlay networks. In *Proc. SOSP*, pages 131–145. ACM Press, 2001.

[3] B. Babcock and C. Olston. Distributed top-k monitoring. In *ACM SIGMODInternational Conference on Management of Data*, pages 28–39, June 2003.

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

[5] D. D. Clark, C. Partridge, J. C. Ramming, and J. Wroclawski. A knowledge plane for the internet. In A. Feldmann, M. Zitterbart, J. Crowcroft, and D. Wetherall, editors, *SIGCOMM*, pages 3–10. ACM, 2003.

[6] `http://comon.cs.princeton.edu/`.

[7] R. Cox, A. Muthitacharoen, and R. T. Morris. Serving DNS using a Peer-to-Peer Lookup Service. In *IPTPS*, 2002.

[8] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 2003.

[9] C. Estan and G. Varghese. New directions in traffic measurement and accounting. In *SIGCOMM*, pages 323–336. ACM, 2002.

[10] M. J. Freedman and D. Mazires. Sloppy Hashing and Self-Organizing Clusters. In *2nd Intl. Workshop on Peer-to-Peer Systems*, 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] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proc. SOSP*, Oct. 2003.

[13] 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.

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

[15] 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.

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

[17] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *Proc. 3rd ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, San Diego, CA, November 2004.

[18] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. PRISM: precision-aware aggregation for scalable monitoring (extended). Technical Report TR-06-22, UT Austin Department of Computer Sciences, May 2006.

[19] D. Kostoulas, D. Psaltoulis, I. Gupta, K. Birman, and A. Demers. Decentralized schemes for size estimation in large and dynamic groups. In *IEEE Network Computing and Applications (NCA 05)*, 2005.

[20] 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.

[21] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *ICDE*, pages 767–778. IEEE Computer Society, 2005.

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

[23] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *VLDB*, pages 144–155, Sept. 2000.

[24] V. Paxson. End-to-end Routing Behavior in the Internet. In *SIGCOMM*, Aug. 1996.

[25] L. Pierce and L. Alvisi. A framework for semantic reasoning about byzantine quorum systems. In *Brief Announcements, Proc. of Symp. on Principles of Distributed Computing*, 2001.

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

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

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

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

[30] 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.

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

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

[33] 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.

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

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

[36] 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.

[37] D. Veitch, S. Babu, and A. Pasztor. Robust synchronization of software clocks across the internet. In *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 219–232, New York, NY, USA, 2004. ACM Press.

[38] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *HotNets-II*, 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] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases to Support Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, Feb. 1999.

[44] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, pages 305–318, 2000.

[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.

8 Appendix

8.1 Arithmetic Imprecision

Mechanism We first describe in detail the aggregation mechanism for a single flow in an aggregation tree for the SUM function with a given AI budget.

8.1.1 Computing SUM for a single attribute

To enforce AI, each aggregation subtree T for an attribute has an error budget δ_T which defines the maximum inaccuracy of any result the subtree will report to its parent for that attribute.

Each node n in the aggregation tree maintains per-attribute state
Ψ_n:$\left\{\delta_{self}, V_{min}, V_{max}, L_{self}, \forall c\big(\delta_c, V^c_{min}, V^c_{max}, L_c\big)\right\}$

Whenever n receives an update from a child c, it triggers the aggregation function that re-computes the aggregate value of all latest received updates from its children.
Function: OnChildUpdate (child c, range [V^c_{min}, V^c_{max}], load L_c)

Step 1. Compute synopses received from children set child(n):

$$P_{max} = \left(\sum_{c \in child(n)} V^c_{max}\right)$$
$$P_{min} = \left(\sum_{c \in child(n)} V^c_{min}\right) \qquad (4)$$

If n has *never* received an update for this attribute from a child c, then [V^c_{min}, V^c_{max}] is set to [0, δ_c].

Step 2. Pass new numeric range through local AI filter:

```
if ($P_{min} < V_{min}$ || $P_{max} > V_{max}$) {
  $V_{min} = P_{min} - bias * \delta_{self}$; // $bias$ ∈ [0, 1];
  $V_{max} = V_{min} + \delta_T$;
  $L_{self}++$;
  $L = \sum_{c \in child(n)} L_c + L_{self}$;
  Send (attr, $Vmax$, $L$) to parent;
}
```

For redistributing the AI budgets in our self-tuning algorithm, M_{self} (M_c) are set to the ratio of L_{self} (L_c for child c respectively) to the time elapsed since the last AI error distribution.

Leaf node: A leaf node can be viewed as an internal node with a single virtual child (the sensor itself) with AI = 0 i.e., the sensor s triggers an update [V_s, V_s] to the leaf node i.e., $V_s = V^s_{max} = V^s_{min}$. Note that the messaging cost of transmitting between the virtual child (sensor) and the leaf node $L_s = 0$ since they reside on the same physical node.

8.1.2 Computing MIN for a single attribute

The mechanism of computing the MIN aggregation function is similar to the SUM where we replace the SUM compuation in Equation 4 by:

$$P_{max} = \left(\min_{c \in child(n)} V^c_{max}\right)$$
$$P_{min} = \left(\min_{c \in child(n)} V^c_{min}\right) \qquad (5)$$

8.1.3 Computing MAX for a single attribute

The MAX aggregation function is symmetric to MIN. Thus, Equation 5 becomes:

$$P_{max} = \left(\max_{c \in child(n)} V^c_{max}\right)$$
$$P_{min} = \left(\max_{c \in child(n)} V^c_{min}\right) \qquad (6)$$

8.1.4 Computing AVG for a single attribute

The AVG aggregation function can be easily computed as a (SUM, COUNT) pair along the same aggregation tree.

8.2 Optimality of Self-tuning AI Error Distribution

The optimal distribution of δ_T among δ_{self} and δ_c is computed as follows: We first find the optimal AI error distribution for a simple degree-2 tree having two levels with the root at level 1 and its two children as the leaf nodes. Later, we will show how this topology can be modeled for any arbitrary d and for any level of the aggregation hierarchy.

Given this topology, we have $\delta_T = \delta_{c1} + \delta_{c2}$; δ_{self} for the root node is set to 0 since it doesn't need to transmit any updates up in the hierarchy. As discussed in Section 3.1, under the assumption that load is inversely proportional to the error budget, we get:

$$\frac{M_{c1}}{M^{opt}_{c1}} = \frac{\delta_{c1}}{\delta^{opt}_{c1}}$$
$$\frac{M_{c2}}{M^{opt}_{c2}} = \frac{\delta_{c2}}{\delta^{opt}_{c2}}$$

We formulate minimizing total load as a multivariate optimization problem sibject to the constraint that the total error budget is fixed i.e.,

$$\text{Minimize} \qquad f : M^{opt}_{c1} + M^{opt}_{c2}$$

subject to the constraint:

$$g : \delta_{c1} + \delta_{c2} - \delta_T = \delta^{opt}_{c1} + \delta^{opt}_{c2} - \delta_T = 0$$

We use Lagrangian multipliers to find the extremum of f($\delta^{opt}_{c1}, \delta^{opt}_{c2}$) subject to the constraint that g($\delta^{opt}_{c1}, \delta^{opt}_{c2}$) =

0 i.e.,

$$\frac{\partial f}{\partial \delta_{c1}^{opt}} + \lambda \frac{\partial g}{\partial \delta_{c1}^{opt}} = 0$$

$$\Rightarrow \quad \frac{\partial}{\partial \delta_{c1}^{opt}} \left(\frac{M_{c1}\delta_{c1}}{\delta_{c1}^{opt}} + \frac{M_{c2}\delta_{c2}}{\delta_T - \delta_{c1}^{opt}} \right) + \lambda \frac{\partial g}{\partial \delta_{c1}^{opt}} = 0$$

$$\Rightarrow \quad -\frac{M_{c1}\delta_{c1}}{\left(\delta_{c1}^{opt}\right)^2} + \frac{M_{c2}\delta_{c2}}{\left(\delta_T - \delta_{c1}^{opt}\right)^2} = 0$$

$$\Rightarrow \quad \frac{\sqrt{M_{c1}\delta_{c1}}}{\sqrt{M_{c2}\delta_{c2}}} = \frac{\delta_{c1}^{opt}}{\delta_T - \delta_{c1}^{opt}}$$

$$\Rightarrow \quad \delta_{c1}^{opt} = \delta_T \left(\frac{\sqrt{M_{c1}\delta_{c1}}}{\sqrt{M_{c1}\delta_{c1}} + \sqrt{M_{c2}\delta_{c2}}} \right)$$

which is a special case of Equation 3 when d = 2. In the general case for a degree-d tree, we get Equation 3:

$$\delta_v^{opt} = \delta_T * \frac{\sqrt{M_v * \delta_v}}{\sum_{v \in \{self\} \cup child(n)} \sqrt{M_v * \delta_v}}.$$

Notes. For an internal-node, we model δ_{self}, M_{self} for that node as a virtual child with $\delta_c = \delta_{self}, M_c = M_{self}$ and use the above equation for $d+1$ children.

8.3 Temporal Imprecision

Pipelined Delays Note that the pipelined delays mechanism significantly reduces the number of updates in an aggregation tree; at each level, an aggregate update is propagated at most once per I seconds where $I = TI - \ell * S$.

To provide the temporal guarantees under synchronized clocks, S must be smaller than $\frac{T}{\ell}$. This implies that temporal imprecision would be violated if $2 * skew_{max} >= (\frac{T}{\ell} - hop_{max}) >= \frac{1}{\ell}(T - \ell * hop_{max})$. The intuition of this result is that $(T - \ell * hop_{max})$ is the total slack for the entire tree; therefore, $skew_{max}$ must be smaller than slack available per level $\frac{1}{\ell}(T - \ell * hop_{max})$.

Proof of Correctness of Pipelined Delays.

Lemma 1. *An event sent by level i at local time T_i is sent by level $i+1$ no later than local time (at level i) $T_i + S$*

Proof. At j^{th} step (TI interval), the extra delay introduced by node at level $i+1$
= $(Z + j * I + (i+1) * S)$ - $(Z + j * I + i * S)$
= S □

Lemma 2. *A leaf event at time X sent by level 0 no later than X + I [True by definition of I]*

Theorem. *An event at a leaf node at local time X is reflected at root no later than time $X + TI$ according to the local time at the same leaf node.*

Proof. An event reaches level d no later than
$X + I + d * S$ [Combining Lemma 1 and Lemma 2]
= $X + (TI - d * S) + d * S$
= $X + TI$ □

In general, if $skew_{max}$ is large due to unsynchronized clocks or weak synchronization we simply fall back on non-pipelined version which we describe next.

Non-pipelined delays. For the case of unsynchronized clocks, we use the same algorithm as the pipelined case with the difference that $2 * skew_{max}$ is no longer used and the parameters Z, I, and S are set slightly differently to reflect lack of coordination between levels. Specifically, $S = hop_{max}$ (we ignore $skew_{max}$) and correspondingly $T = \ell * (I + S)$. Note that we get a different bound on minimum temporal imprecision as $T_{min} = \ell * hop_{max}$ for the non-pipelined case.

In terms of implementation, instead of a global reference time as in the synchronized case, the aggregation function specifies an arbitrary reference time Z. Therefore, at local time $Z + i * I$ corresponding to a node at any level of the aggregation tree, it sends an updated aggregate value to its parent iff the value of any of its inputs has changed since time $Z + (i-1) * I$. The efficiency for non-pipelined case is qualitatively same as the pipelined case—at each level, an aggregate update is propagated at most once per I seconds.

Proof of Correctness of Non-Pipelined Delays. To prove correctness, we define the following lemma:

Lemma 3. *At any level, the maximum delay in update propagation by a node is $I + S$.*

This leads to the proof of Theorem 8.3 for the non-pipelined case.

Proof. An event reaches level d no later than $X + d * (I + S)$ = X + TI □

Comparison. Given the same *a priori* temporal imprecision budget, the value of I for the pipelined and the non-pipelined cases would be different i.e.,

$$\begin{aligned} I_{pipelined} &= T - \ell * S \\ &= \ell * \Big(\frac{T}{\ell} - hop_{max} - 2 * skew_{max} \Big) \\ I_{non\_pipelined} &= \frac{T}{\ell} - hop_{max} \end{aligned}$$

Therefore, when skew is small, i.e. $2 * skew_{max} \ll \frac{T}{\ell} - hop_{max}$, pipelined delay can achieve almost a factor of ℓ reduction in the update frequency.

8.4 Network Imprecision

Here we present the analytical results for computing the expected NI using k aggregation trees after f independent failures have occured. Note that by ”f independent failures”, we allow two failures to be on the same node; in this case, their contribution to NI is counted twice.

Notation.

- c: the number of logical children a node has in the aggregation tree (i.e., logical fanout).
- d: the depth of the aggregation tree
- P(i,f,k): with k random trees, the probability for at least one tree to have all failures occuring at level $<= i$ (leaf at level 0) (which implies the NI = $N_{all} - N_{reachable} <= f * c^i$ with f independent failures because each node at level i contribute at most c^i to NI).
- E(NI,f,i): the expected NI with f independent failures conditioned on the fact every failure appears in max level of i or below
- Var(NI,f,i): the variance of NI with f independent failures conditioned on the fact every failure appears in max level of i or below.

8.4.1 Tail Probability Analysis

The probability for a failure to appear in max level ¡= i (leaf is level 0) i.e., Pr(failure in max level $<= i$) = 1 - Pr(failure in max level $> i$)
= $(1 - 1/c^{i+1})$.

The probability for all f independent failures to appear in level $<= i$ is therefore $(1 - 1/c^{i+1})^f$. Note that the contribution to NI by each failure with max level $<= i$ is at most c^i.

Therefore we get P(i,f,1) = $(1 - 1/c^{i+1})^f$.

With k random aggregation trees, the probability for at least one tree to have NI ¡= $f * c^i$ is

$$P(i,f,k) = 1-(1-P(i,f,1))^k = 1-[1-(1-1/c^{i+1})^f]^k \tag{7}$$

Example. Suppose c = 8, f = 10, k = 4, then

a. with prob. $>=$ 99.95% $N_{all} - N_{reachable} <= f * c^1$ = 80

b. with prob. $>=$ 70.51% $N_{all} - N_{reachable} <= f * c^0$ = 10

Analysis on the Expected NI. We will use the above analysis to prove that with high probability, at least one aggregation tree has every failure appearing at max level $<= i$.

We first analyze the mean and standard deviation for NI conditioned on the fact that all failures occur at max level $<= i$. For a single failure, the probability for the failure to occur at max level j (conditioned on the fact its max level is $<= i$) is

$$Q(j) = \frac{1/c^j - 1/c^{j+1}}{1 - 1/c^{i+1}}$$

Its contribution to NI is exactly c^j. Therefore, with 1 failure, we have

$$\begin{aligned} E(NI,1,i) &= \sum_{j=0\ldots i} Q(j) * c^j \\ &= (i+1) * \frac{(1-1/c)}{(1-1/c^{i+1})} \\ Var(NI,1,i) &= E(NI^2) - E(NI)^2 \\ &= \sum_{j=0\ldots i} Q(j) * c^{2*j} - E(NI)^2 \\ &= c^i - E(NI)^2 \end{aligned}$$

With f independent failures, we get: E(NI,f,i) $<= f * E(NI,1,i)$ and Var(NI,f,i) $<= f^2$*var(NI,1,i)

Note that we use ”$<=$” instead of ”$=$” because we ignore the fact that one failure may be the ancestor of another.

Example. Suppose c = 8, f = 10, k = 4, i = 1, then with prob. $>=$ 99.95% $N_{all} - N_{reachable} <= f * c^1$ = 80 and in this case, E(NI,f,i) $<=$ 17.78 and STDDEV.(NI,f,i) $<=$ 22.

To summarize, a fundamental and unique challenge inherent in any hierarchical aggregation system (regardless of whether DHT is used) is the failure amplification effect – once a non-leaf node fails, an entire subtree rooted at that node is affected. With d levels, with f failures, although the expected number of affected nodes is only $(d+1) * f$, the standard deviation is really high = $f * c^{d/2}$. As a result, with fairly high probability, f failures can be amplified to a much larger number of affected nodes even when f is very small.

The metrics we proposed in Section 4

a. quantify the amplification effect for each aggregation tree. In particular, since our dual-tree aggregation scheme allows us to monitor the NI for every tree, by taking the mean of all NIs, we can accurately estimate $(d+1) * f$ because the standard deviation is now $f * c^{d/2}/c^d = f/c^{d/2}$. If we normalize the NI for an aggregation tree by $(d+1) * f$, we then get the amplification factor for that tree.

b. give us a way of reducing amplification effect by using multiple trees for the same aggregation function.