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. Exposing imprecision is essential for both correctness in the face of network and node failures and for scalability to large systems. PRISM quantifies imprecision along a three-dimensional vector: *arithmetic imprecision* (AI) and *temporal imprecision* (TI) balance precision against monitoring overhead while *network imprecision* (NI) addresses the challenge of providing consistency guarantees despite failures. Our implementation provides these metrics in a scalable way via (1) self-tuning of AI budgets to shift imprecision to where it is useful, (2) pipelining of TI delays to maximize batching of updates, and (3) dual-tree prefix aggregation which exploits regularities in our DHT's topology to drastically reduce the cost of the active probing needed to maintain NI. PRISM's careful management of imprecision qualitatively improves its capabilities. For example, by introducing a 10% AI, PRISM's PlanetLab monitoring service reduces network overheads by an order of magnitude compared to PlanetLab's CoMon service, and by using NI metrics to automatically select the best aggregation results, PRISM reduces the observed worst-case inaccuracy of our measurements by nearly a factor of five.

1 Introduction

Scalable system monitoring is a fundamental abstraction for large-scale networked systems. It serves as a basic building block for applications such as network monitoring and management [9, 23, 51], contractible monitors [29], resource location [26, 50], efficient multicast [48], sensor networks [26, 50], resource management [50], and bandwidth provisioning [15]. To provide a real-time view of global system state for these distributed applications, the central challenge for a monitoring system is scaling to keep track of thousands or millions of dynamic attributes (e.g., per-flow or per-object state) spanning tens of thousands of nodes.

Recent work on aggregation [26, 33, 48, 50] and DHTs [8, 38, 40, 41, 45, 56] seeks to provide monitoring at such scale. However, to fully realize the goal of scalable system monitoring, the underlying monitoring infrastructure must expose imprecision in a controlled manner for two reasons.

First, introducing controlled amounts of *arithmetic imprecision* (AI) and *temporal imprecision* (TI) can reduce monitoring load by an order of magnitude or more for some applications. Studies suggest [30, 34, 44, 48, 54] that real-world applications often can tolerate some inaccuracy as long as the maximum error is bounded and that small amounts of imprecision can provide substantial bandwidth reductions.

Second, a monitoring service that fails to expose *network imprecision* (NI) due to failed/slow nodes and network paths risks delivering arbitrarily incorrect results. In particular, failures and reconfigurations can prevent nodes in a large-scale system from delivering important updates, which makes it fundamentally difficult to ensure that a reported value corresponds to the system's actual state [19]. This problem is heightened in a hierarchical aggregation system because of the *failure amplification effect* [33]: 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.

To address these needs, we have developed PRecision-Integrated Scalable Monitoring (PRISM). PRISM builds on recent work that uses DHTs to construct scalable, load-balanced forests of self-organizing aggregation trees [8, 17, 39, 42, 50]. However, to realize the vision of scalable monitoring, PRISM should address two key technical challenges to provide a controlled tradeoff between imprecision and load. First, although it is relatively straightforward to reduce load by filtering some updates, doing so while ensuring bounds on the accuracy of the reported results is difficult. Second, PRISM must provide implementations of AI, TI, and NI that scale to tens of thousands of nodes and millions of attributes. PRISM meets these challenges using four novel techniques:

- The central principle guiding the design of PRISM is the notion of *conditioned consistency*: the AI and TI results 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. Conditioned consistency is vital because it (1) simplifies our implementations of AI and TI because they can ignore the difficult corner cases that arise due to failures and (2) provides a clean way to address the fundamental problem of network churn distorting monitoring results.
- For AI, PRISM employs a hierarchical self-tuning algorithm that directs imprecision slack to where it is most needed. Self-tuning distribution of AI budgets can reduce monitoring costs by more than a factor of two over static uniform distribution.
- For TI, PRISM pipelines the available slack across levels of the aggregation hierarchy to maximize the number of updates batched together. Batching reduces monitoring load by an order of magnitude for some workloads.
- For NI, PRISM introduces a novel *dual-tree prefix aggregation* that exploits symmetry in our DHT-based aggre-

gation topology to reduce NI monitoring overhead by orders of magnitude. Specifically, NI must track how many nodes' current updates are reflected in each aggregation tree in our scalable DHT-based system. This tracking is difficult because (a) any failed node or link potentially affects different aggregation trees in different ways and (b) detecting failures in the presence of AI caching and TI constraints requires frequent active probing of nodes. By using dual-tree prefix aggregation, PRISM reduces the per-node overhead of tracking NI from $O(n)$ to $O(\log n)$ messages per second in an n-node system.

Experience with a Distributed Heavy Hitter detection (DHH) application, a Ulim library for resource isolation between distributed services, 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 PrMon and DHH 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 [10] 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 a scalable 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 incur unacceptable overheads.

2 Background

PRISM builds on two ongoing research efforts for scalable monitoring: aggregation and DHT-based aggregation.

Aggregation is a fundamental abstraction for scalable monitoring [8, 17, 26, 38, 48, 50] because it allows applications to access summary views of global information and detailed views of rare events and nearby information.

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. 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) [22, 50]. 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.

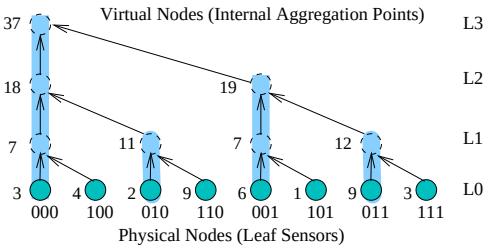


Fig. 1: The aggregation tree for key 000 in an eight node system. Also shown are the aggregate values for a simple SUM() aggregation function.

The tree-based aggregation in the PRISM framework is defined in terms of an aggregation function 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 an aggregation tree, PRISM defines an *aggregate value* $V_{i,attr}$ for each attribute: 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. The aggregate value for a level-i subtree T_i is the result returned by the aggregation function computed across the aggregate values of T_i's children. Figure 1, for example, illustrates the computation of a simple SUM aggregate.

DHT-based aggregation. PRISM leverages DHTs [38, 40, 41, 45, 56] to construct a forest of aggregation trees and maps different attributes to different trees [8, 17, 38, 42, 50] 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 corresponding 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 [8, 17, 38, 42, 50].

3 Example Applications

Aggregation is a building block for many distributed applications such as network management [51], service placement [18], sensor monitoring and control [30], multicast tree construction [48], and naming and request routing [12]. In this paper, we focus on three case-study examples: a distributed heavy hitter detection, a distributed monitoring service for PlanetLab modeled on CoMon [10], and a "Ulim" library for resource isolation between distributed services.

Heavy Hitter detection: 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 [15]. The key challenge

for this distributed query is scalability for aggregating per-flow statistics for millions of concurrent flows in real-time. For example, the Abilene [2] 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 feed into the second. First, PRISM calculates the total incoming traffic for each destination 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.

Real-time Network Monitoring: The second application is our PrMon monitoring service that is representative of monitoring Internet-scale distributed systems such as PlanetLab [37] and Grid systems [47] 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 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. Although there exist other centralized monitoring services, in Section 5 we show that PRISM can monitor a large number of attributes at much finer time scales while incurring significantly lower network costs.

Ulim Library: The final application we implement in the PRISM framework is an application-level "Ulim library" that monitors and enforces the global resource usage of a distributed service or experiment deployed on a large-scale networked system e.g., PlanetLab [37]. The goal is to provide a distributed bounding box that increases resource isolation between different services, thereby safeguarding the system against anomalies such as denial of service attacks and buggy experiments [5]. A user attaches her experiment to Ulim and specifies a resource usage policy describing its expected traffic rates, CPU requirements, memory usage, etc. The Ulim library leverages PRISM monitoring to enforce this policy across the system; if an experiment exceeds its pre-specified resource limits, then (1) the user gets an email notification of the policy violation and (2) the experiment is terminated using a user-provided security key. We further envision that the Ulim framework will provide a key building block for specification-based intrusion detection in future networks like GENI [4].

4 PRISM Design

In this section we present the system design and describe how to enforce imprecision limits and quantify the consistency guarantees in PRISM. PRISM's core architecture is a DHT-based aggregation system that achieves scalability by mapping different attributes to different aggregation trees [8, 17, 38, 42, 50]. PRISM then introduces controlled tradeoffs between precision guarantees and load.

4.1 Overview

PRISM quantifies imprecision along a three-dimensional vector: (Arithmetic, Temporal, Network). *Arithmetic imprecision* (AI) bounds the numeric difference between a reported value of an attribute and its true value [35, 55], and *temporal imprecision* (TI), bounds the delay from when an update is input at a leaf sensor until the effects of the update are reflected in the root aggregate [44, 55]. 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.

Network imprecision (NI) characterizes the uncertainty introduced by node crashes, slow network paths, unreachable nodes, and DHT topology reconfigurations. If these issues are not addressed by a monitoring system, the results it reports may be *arbitrarily incorrect.* Unfortunately, ensuring that a reported value reflects all recent updates is fundamentally hard [19], and coping with these sources of error is particularly challenging for PRISM for three reasons. First, because PRISM uses AI caching, if a subtree is silent over an interval, PRISM must distinguish two cases: (a) the subtree has sent no updates because the inputs have not significantly changed from the cached AI values or (b) the inputs have significantly changed but the subtree is unable to transmit its report. Second, because PRISM uses TI to batch updates, there are windows of time in which a short disruption can block a large batch of updates and greatly perturb the system's state and outputs. Third, in any hierarchical aggregation system the problem of dealing with failures is made worse by the *amplification effect*: 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.

The key idea of NI is that because no system can guarantee to always provide the "right" answer [19, 43], 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 comprises 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 switch to a new parent while its old parent retains the node's inputs as soft state until a timeout.

PRISM's design explicitly separates the mechanism for detecting and quantifying NI via these metrics from the policy question of how to minimize the damage caused by network and node failures. In particular, different applications will react to NI values according to their different requirements. We discuss several examples of how applications use NI in Section 4.4.4.

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 $[V_{min}, V_{max}, \textit{TI}, N_{all}, N_{reachable}, N_{dup}]$ 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."

Integrating AI, TI, and NI is central to PRISM's design. The AI and TI implementations are simple because they can assume that aggregation trees never reconfigure and that nodes and network paths never fail and are never slow despite the long tail of Internet RTTs [3, 13, 36]. The NI metric then addresses these challenging real-world issues.

4.2 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.

4.2.1 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 (with $\delta_T \geq \delta_{self} + \sum_{c \in children} \delta_c$), 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 similar [1].

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 $V_{max} \leq V_{min} + \delta_T$), 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

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

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

Note the trade-off in splitting δ_T between δ_{self} and δ_c. Large δ_c allows 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 *distribution 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}.

4.2.2 Self-tuning error budgets

The key AI policy question is how to divide a given error budget δ_{root} across the nodes in an aggregation tree.

A simple approach is a static policy that divides the error budget uniformly among all the children. E.g., 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.

To make cost/accuracy tradeoffs *self-tuning*, PRISM provides an adaptive algorithm. 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. Limiting redistribution overhead is a particular concern for applications like distributed heavy hitter that monitor 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 across 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 messages 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^{opt}_v that minimizes the total system load $\sum_v M^{opt}_v$, where M^{opt}_v is an estimate of the load generated by node v under optimal error budget δ^{opt}_v. In particular, for any $v \in \{self\} \cup$

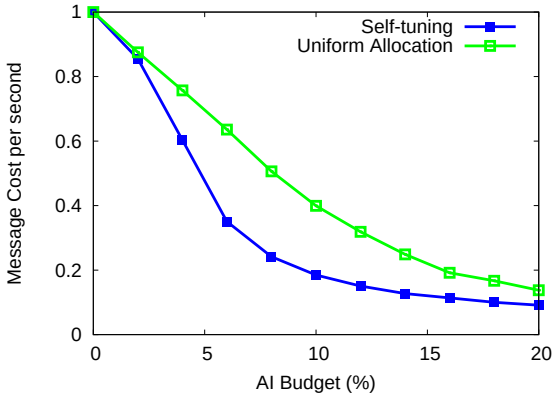


Fig. 2: Self-tuning vs. uniform static AI error distribution using simulator.

$child(n)$ we estimate

$$\delta_v^{opt} = \delta_T * \frac{\sqrt{M_v * \delta_v}}{\sum_{v \in \{self\} \cup child(n)} \sqrt{M_v * \delta_v}} \quad (2)$$

which is optimal [1] assuming that load is inversely proportional to error budget. However, not all workloads exhibit this property. Nonetheless, this heuristic seems reasonable for estimating the impact of small changes in δ for a range of workloads, and we find that it works well in practice.

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 when doing so is likely to save at least k messages (i.e., if $charge_c > k$). To ensure the invariant that $\delta_T \geq \delta_{self} + \sum_c \delta_c$, 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, $\delta_c^{opt} - \delta_c$, max(0.1$\delta_{self}$, δ_{self} - δ_{self}^{opt})). Second, we replenish δ_{self} from the child whose δ_c is the farthest above δ_c^{opt} by ordering c to reduce δ_c by min(0.1 δ_c, δ_c - δ_c^{opt}).

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

Figure 2 shows the communication cost for the self-tuning algorithm compared to uniform static distribution of AI budgets using a simulator for 256 leaf nodes in a 4 level degree-4 tree, accounting for the redistribution overhead. We generate the leaf data values using a random walk model, using randomly assigned step size. We observe that self-tuning reduces overhead by more than a factor of two for this workload.

4.2.3 Implementation details

Given these mechanisms, we still have plenty of freedom to (i) set δ_{root} to an appropriate value for each attribute, and (ii) compute V_{min} and V_{max} when updating a parent.

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 then distributed down the tree via a distribution function. We have prototyped systems that use each of these three policies.

Computing [V_{min}, V_{max}]. When either $\sum_c V_{min}^c$ or $\sum_c V_{max}^c$ goes outside of the last [V_{min}, V_{max}] that was reported to the parent, a node needs to report a new range. 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 $V_{min} = (\sum_c V_{min}^c) - bias * \delta_{self}$ and $V_{max} = (\sum_c V_{max}^c) + (1 - bias) * \delta_{self}$ to the parent. The *bias* parameter can be set as follows:

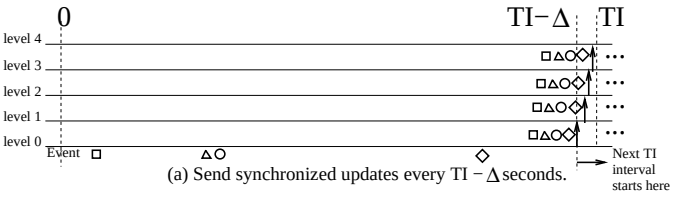
- $bias \approx 0.5$ if inputs expected to be roughly stationary
- $bias \approx 0$ if inputs expected to be generally increasing
- $bias \approx 1$ if inputs expected to be generally decreasing

For example, suppose a node with total δ_T of 10 and δ_{self} of 3 has two children reporting ($[V_{min}^c, V_{max}^c]$) 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.

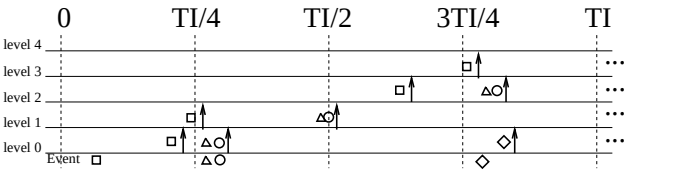
4.3 Temporal Imprecision

Temporal imprecision provides a bound on the delay from when an update occurs at a leaf node to 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.

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. Second, explicitly exposing TI provides an opportunity to combine multiple updates to improve scalability by



(a) Send synchronized updates every TI − Δ seconds.



(b) Send unsynchronized updates every TI/ *l* seconds.

Fig. 3: For a given TI bound, pipelined delays with synchronized clocks (a) allows nodes to send less frequently than unpipelined delays without synchronized clocks (b).

reducing processing and network load.

To maximize the possibility of batching updates, when clocks are synchronized[1], we pipeline delays as shown in Figure 3(a) so that each node sends once every $(TI - \Delta)$ seconds with each level's sending time staggered so that the updates from level i arrive just before level $i + 1$ can send. The extended technical report [1] details how we set each level's sending time while coping with transmission delays and clock skew across nodes. As detailed there, accounting for the worst case delays hop_{max} and skews $skew_{max}$ yields $\Delta = \ell * (hop_{max} + 2 * skew_{max})$, and it guarantees the following property: an event at a leaf node at local time X is reflected at the root no later than time $(X + TI)$ according to the local time at the leaf node.

Conversely, if clocks are not synchronized, then we fall back on a simple but less efficient approach of having each node send updates to its parents once per TI/ℓ seconds as illustrated in Figure 3(b).

4.4 Network Imprecision

In this section we describe how PRISM provides a scalable implementation of the NI metrics and how applications use these metrics to interpret global aggregate results. It is important to note that whereas AI and TI are calculated and enforced on a per-attribute basis, NI is maintained by the system for each aggregation tree and shared across all attributes mapped to each tree. This arrangement both amortizes the cost of maintaining NI and simplifies the definition of attributes' aggregation functions.

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 result might include reports of disconnected subtrees as well as double count the contribution of rejoined subtrees.econd, 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 4.4.3.

[1]Algorithms in the literature can achieve clock synchronization among nodes to within one millisecond [49].

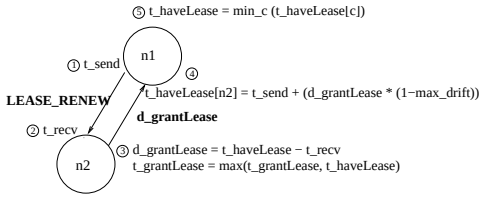


Fig. 4: Protocol for a parent to renew a lease on the right to hold as soft state a child's contribution to an aggregate.

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 4.4.2 we explain how PRISM computes N_{dup} to account for dynamically changing aggregation topologies. Later, in Section 4.4.3 we describe how to scale the approach to work with the large number of distinct trees constructed by PRISM's DHT framework. Finally, Section 4.4.4 describes how NI characterizes the "completeness" of results and how applications can use this information in different ways.

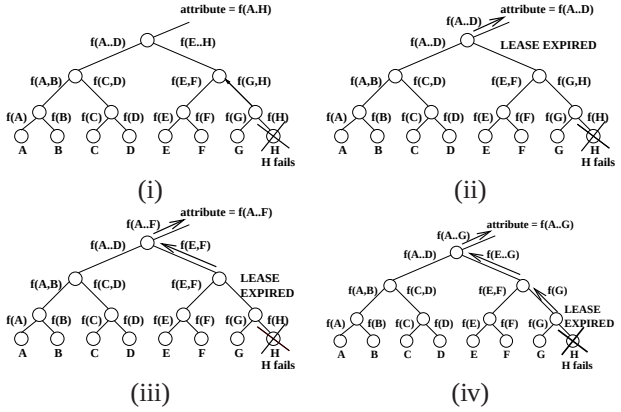
4.4.1 Single tree, static topology

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

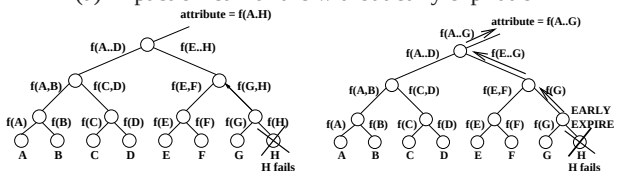
N_{all} is simply a count of all nodes in the system, which serves as a baseline for evaluating $N_{reachable}$ and N_{dup}. N_{all} is easily computed using PRISM's aggregation abstraction. Each leaf node inserts 1 to the N_{all} aggregate, which has SUM as its aggregation function. Note that even if a node becomes disconnected from the DHT, its contribution to 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}. Recall that TI is calculated *assuming* good connectivity and bounds on message delay. $N_{all} - N_{reachable}$ represents the number of nodes whose inputs may fail to meet these assumptions. 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 is not responsive, 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 [1].



(a) Impact of leaf failure without early expiration



(b) Impact of leaf failure with early expiration

Fig. 5: Recalculation of aggregate function across values A, B, ..., H after the node with input H fails (a) without and (b) with early expiration.

Nreachable v. TI The difference between N_{all} and $N_{reachable}$ characterizes the count of nodes that may currently be violating their TI bounds. However, past connectivity disruptions could affect attributes with large TI. In particular, to maximize batching, our TI algorithm defines a small window of time during which a node must propagate updates to its parents, so any attribute's subtree that was unreachable over the last TI_{attr} could have been unlucky and missed its window even though the subtree 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) 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.

4.4.2 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 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" [46]), we use long timeouts to avoid spurious garbage collection (e.g., we use $T_{declareDead} \approx 10$ minutes in our prototype.)

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. Note that our implementation also allows a user to define duplicate-insensitive aggregation functions where possible [11, 33]. However, to support a broader range of aggregation functions, PRISM computes N_{dup} for each aggregation tree. 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: 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 its input of N_{dup} to 0.

Our 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 disruption when nodes near the leaves of a tree fail or move.

Lease aggregation. For correctness, our implementation uses a *lease aggregation* algorithm that extends the concept of leases [20] to hierarchical aggregation.

Figure 4 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 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 [52].

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.

Early expiration. PRISM uses *early expiration* to minimize the scope of disruption when a tree's topology reconfig-

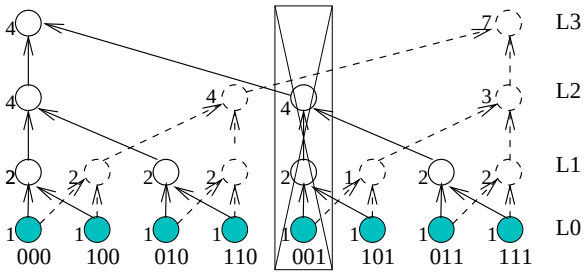


Fig. 6: The failure of a physical node has different effects on different aggregations depending on which virtual nodes are mapped to the failed physical node. The numbers next to virtual nodes show the value of $N_{reachable}$ for each subtree after the failure of physical node 001, which acts as a leaf for one tree but as a level-2 subtree root for another.

ures. 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 5(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 5 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 and bring into play the amplification effect.

Early expiration avoids this unwarranted disruption as Figure 5(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.

4.4.3 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 underlying DHT and then uses different aggregation trees for different attributes. As Figure 6 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 4.4.1 explained, maintaining the NI metrics requires frequent active probing along each edge of each tree's graph.

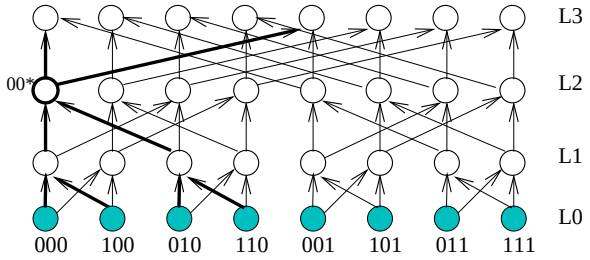


Fig. 7: Plaxton tree topology is an approximate butterfly network. The bold connections illustrate how a virtual node 00* uses the dual tree prefix aggregation abstraction to aggregate values from a tree below it and distribute the results up a tree above it.

As 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), have total $\Theta(n^2)$ edges that must be monitored; such monitoring would require $\Theta(n)$ messages per node every probe interval (p = 10s in our prototype). To put this in perspective, consider a $n = 512$-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 50 probes per second. As the system grows, the situation deteriorates rapidly—a 1024-node system requires each node to send roughly 100 probes per second.

Our solution, described below, reduces active monitoring work to $\Theta(d \log n)$ probes per node per p seconds. The 512-node system in the example will require each node to send about 5 probes per second; the 1024-node system will require each node to send about 5.8 probes 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 [38] to re-use common sub-calculations across different aggregation trees using a novel *dual tree prefix aggregation* abstraction.

As Figure 7 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 7 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 7, 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 4 bits per hop and d=16, so each subtree is common to 16 parents.

4.4.4 Using NI

PRISM's formulation of NI explicitly separates the basic mechanism for detecting and quantifying NI from the policy for dealing with it. 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 severity of the disruption (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 [11]), 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, given the broad range of factors that determine the significance of NI disruptions, 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 metrics is nonetheless powerful—it supports a broad range of techniques for coping with NI from network and node disruptions. Examples include

- *Filtering or flagging unacceptably uncertain answers*—systems can manage the trade-off between consistency and availability [19] by sacrificing availability (e.g., throwing an exception rather than returning an answer when NI exceeds some threshold). Conversely, a system could maximize availability by always returning an answer based on the best available information but flagging that answer's quality as high, medium, or low depending on the current NI.
- *Increasing reported TI*—short bursts of low $N_{reachable}$ mean that an aggregated value may not reflect some recent updates. Rather than report a "low quality" result for the current period, a system can report a "high quality" result with explicitly less temporal precision.
- *On-demand reaggregation*—given a signal that current results may be missing updates from some sensors, a system can trigger a full on-demand reaggregation to gather current reports (without AI caching or TI buffering) from whatever sensors are available.
- *Duplicate-insensitive aggregation*—some systems can be designed with duplicate-insensitive aggregation functions. For example, MAX is inherently duplicate-insensitive [30], and duplicate-insensitive approximations of some other functions exist [11, 31, 33].
- *Redundant aggregation*—systems can aggregate an attribute using several different keys so that one of the keys is likely to find a route around the disruption. Our theoretical analysis shows that under f independent failures in an ℓ-level d-ary aggregation tree, the expected number of disconnected nodes is $f * (\ell + 1)$ with high standard deviation $f * d^{\frac{\ell}{2}}$. However, by aggregating an attribute along a (small) constant number of trees, all failure occur at level $\leq i$ ($i << \ell$) with mean $f * (i + 1)$ and standard deviation $f * d^{\frac{i}{2}}$ with high probability. For example, for ℓ = 2, d = 16, f = 10, $i = 1$, aggregating an attribute along 4 trees decreases deviation from 160 for a single tree to 40; detailed proofs are in the technical report [1]. Later in section 5.4, we show that by aggregating an attribute up k paths and using the answer corresponding to the path with the lowest overall NI disruption, we can reduce inaccuracy by nearly a factor of five for $k = 4$.

These examples illustrate how to use NI. More generally, given information about the quality of a reported answer, different applications can take different actions to cope with network disruptions.

5 Experimental Evaluation

We have developed a prototype of the PRISM monitoring system on top of FreePastry [41].

Our experiments characterize the performance and scalability of the AI, TI, and NI metrics. First, we 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 trade-offs 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 two applications: the PRISM Monitoring service PrMon and Distributed Heavy Hitter (DHH).

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 gathered CoTop [10] data from 200 PlanetLab nodes at 1-second intervals for 1 hour. The CoTop data provide the per-slice usage of 9 CPU, NW, and memory resources for all slices running on each node. Using these logs as sensor input, we run PRISM on 200 servers mapped to 50 Emulab machines. Note that for comparison with CoMon, the baseline is set to AI of -1 (no AI caching) and TI of 5 minutes.

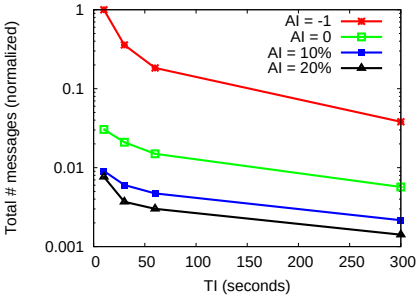


Fig. 8: Load vs. AI and TI for PrMon application.

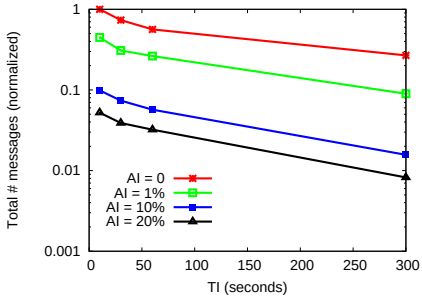


Fig. 9: Load vs. AI and TI for DHH application.

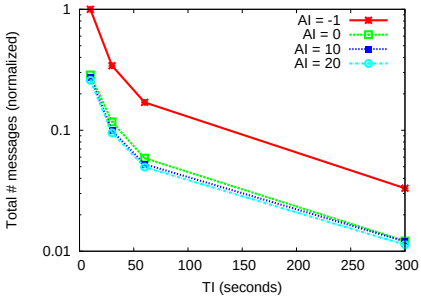


Fig. 10: Load vs. AI and TI for PrMon's CPU attribute

Figure 8 shows the combined effect of AI and TI in reducing PrMon's load for monitoring all the running PlanetLab slices in our CoTop trace data. The x-axis shows the TI budget and the y-axis shows the total message load during the 1-hour run normalized with respect to AI of -1 and TI = 10 seconds. 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; the corresponding message overhead per node is about 90 messages per second (TI = 10s) and 4 messages per second (TI = 5 minutes). Likewise, for a fixed TI of 10 seconds, AI of 20% reduces load by two orders of magnitude (to 0.7 messages per node per second) compared to AI = -1. By combining AI of 20% and TI of 30 seconds, we get both an order of magnitude load reduction (to 0.3 messages per node per second) 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 DHH application, we use multiple netflow traces obtained from the Abilene [2] Internet2 backbone network. The data was collected from 3 Abilene routers for 1 hour; each router logged per-flow data every 5 minutes, and we split these logs into 400 buckets based on the hash of source IP. As described in Section 3, our DHH application 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 15 second moving window shifted every 5 seconds.

Figure 9 shows the precision-performance results for the top-10 DHH query for 400 nodes mapped to 100 Emulab machines. The total monitoring load is normalized relative to the load for AI of 0 and TI of 10 seconds. The AI budget is varied from 1% to 20% of the maximum flow's global traffic volume. We observe that AI of 10% reduces monitoring load by an order of magnitude compared to AI of 0 for a fixed TI of 10 seconds, by (a) culling all updates for large numbers of "mice" flows whose total bandwidth is less than this value and (b) filtering small changes in the remaining elephant flows. Similarly, TI of 5 minutes reduces load by about 80% compared to TI of 10 seconds. For DHH application, AI filtering is more effective than TI batching for reducing load because of the large fraction of mice flows in Abilene traces.

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

5.2 Setting Monitoring Budget

Finally, to reduce bandwidth, one can either increase AI or TI. We provide two guidelines: (1) for attributes that exhibit large variation in consecutive updates (e.g., CPU), bandwidth falls roughly proportionally with increasing TI but increasing AI may have little impact because it most updates will bypass AI filtering under modest error budgets as shown in Figure 10 and (2) for attributes that show small variance (e.g., number of processes), increasing the AI budget may be effective.

5.3 Promised vs. Realized Accuracy

A central goal of PRISM is to go beyond providing best effort imprecision estimates to ensuring worst-case guarantees conditioned by NI. In this subsection, we experimentally investigate PRISM's accuracy by using the CoTop trace for the "CPU" attribute, configuring PRISM with different AI and TI values, playing that trace through PRISM on 200 servers mapped to 50 Emulab nodes, logging the value reported for the attribute at each second, and doing an off-line comparison between the PRISM's reported values and trace inputs.

First, we experimentally test whether the results delivered by PRISM do, in fact, remain within the range promised by PRISM's imprecision guarantees. We compare PRISM's actual output at every second to the *oracle* output computed across the input traces for AI values of 0, 1%, 5%, and 10% with TI values of 1s and 10s. In 99.9% (3596 of 3600) of the 1-second periods at the various levels of AI and TI, the reported value lies within the range promised by PRISM; the inaccuracy of less than 1% in the remaining 0.1% of reports stems from disruptions captured by the NI metrics as we discuss in detail in Section 5.4.

Next, we examine how different levels of AI and TI affect the actual end-to-end imprecision delivered to applications relative to the instantaneous oracle value computed across the input traces. Figure 11 and 12 show for different values of AI, the CDF of deviation between PRISM's reports compared to the oracle truth for fixed TI of 1s and 10s, re-

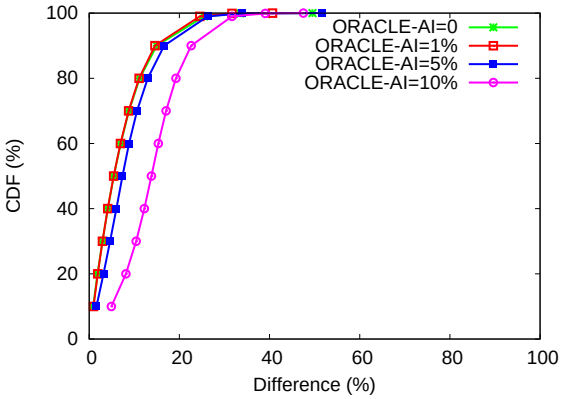


Fig. 11: CDF for difference between PRISM's reported values and oracle truth for fixed TI of 1 second.

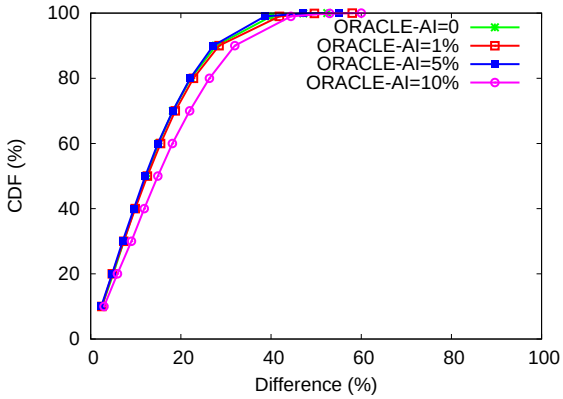


Fig. 12: CDF for difference between PRISM's reported values and oracle truth for fixed TI of 10 seconds.

spectively. We make two observations here: First, for AI of 5% and 1 second TI, more than 90% of reports have less than 16% difference from the oracle. Notice, however, that even with AI of 0 and immediate propagation, any aggregation system's reports can differ from the oracle truth due to propagation delays. As illustrated in Figure 12, increasing the TI to 10 seconds results in a larger deviation between PRISM's reported results and the oracle. Second, for AI of 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.

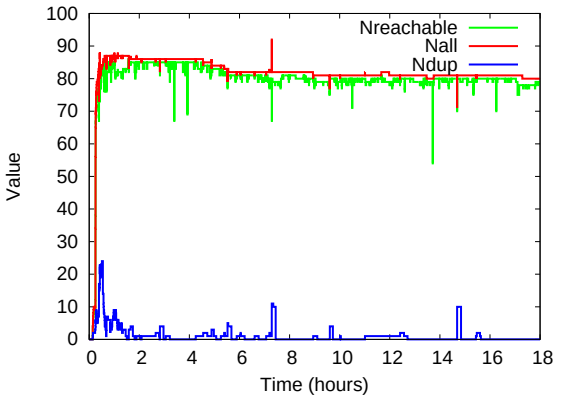


Fig. 13: NI metrics reflecting PlanetLab state (85 nodes).

5.4 NI: Coping With Disruption

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

Whereas Sections 5.1 and 5.3 show system performance under stable conditions (low NI), in the rest of this section we focus on NI's effectiveness during periods of instability. In particular, we run experiments on PlanetLab nodes. Because these nodes 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 a convenient stress test of our system.

Figure 13 shows how NI reflects network state for a 85-node PlanetLab experiment for a 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 to 80.

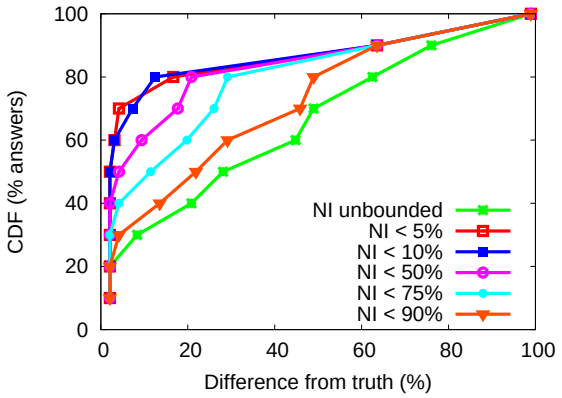


Fig. 14: CDF for reported answers filtered for different NI thresholds and K = 1.

5.5 Consistency v. Availability

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 14 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. Conversely, for monitoring systems that ignore NI (*no filtering* line), half of their reports differ from the truth by more than 60%. As discussed in Section 4.4.4, applications can filter results using different NI thresholds and take an appropriate action to correct distorted results.

In Figure 15 we explore the effectiveness of redundant aggregation (Section 4.4.4) i.e., using K redundant trees to compute an attribute and then using NI to identify the highest-quality result. Figure 15 shows the CDF of results with respect to the deviation from 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 AI and TI to increase K and thereby greatly reduce NI.

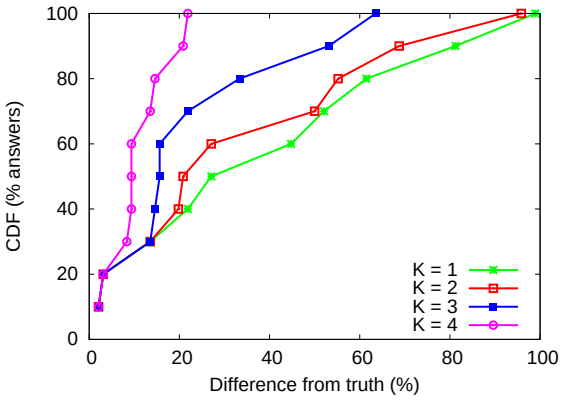


Fig. 15: CDF of NI values for different K.

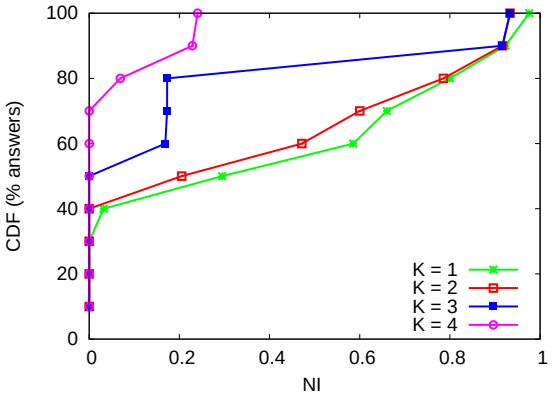


Fig. 16: CDF of NI values for K duplicate keys.

Filtering answers during periods of high churn exposes a fundamental consistency versus availability tradeoff [19]. Figure 16 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 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 [1].

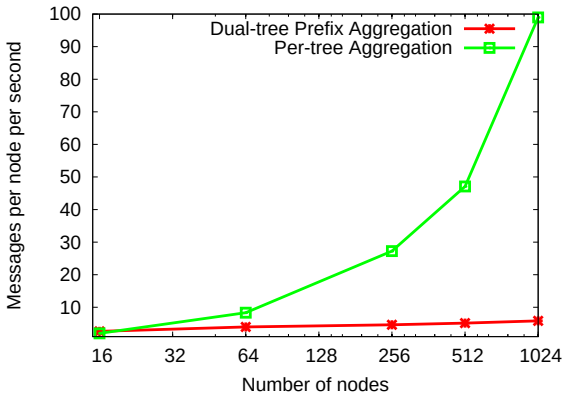


Fig. 17: NI monitoring overhead for dual-tree prefix aggregation compared to computing NI per aggregation tree; x-axis is on a log scale.

5.6 NI Scalability

Finally, we empirically quantify the monitoring overhead of tracking NI via (1) each aggregation tree and (2) dual-tree prefix aggregation. Figure 17 shows the 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.

6 Related Work

Aggregation systems commonly use some form of AI or TI to reduce monitoring overheads. Olston et al. [6, 34] use adaptive filters at the data sources that compute approximate answers for continuous queries in single-level communications topologies. Manjhi et al. [32] determine an optimal but *static* distribution of slack to the internal and leaf nodes of an tree for the special case of finding frequent items in database streams. In comparison, PRISM supports general aggregation functions and employs a self-tuning algorithm for distributing the error budgets in a general communication hierarchy. IrisNet [14] filters sensors at leaves and caches timestamped results in a hierarchy with queries that specify the maximum staleness they will accept and that trigger re-transmission if needed. In contrast, PRISM coordinates transmission of push-based continuous query results to support in-network aggregation, allowing it to more aggressively optimize the TI batching. TAG [30] bounds TI by partitioning time into intervals of duration $\frac{TI}{l}$ (l: maximum tree level) with nodes at level i transmitting during the i^{th} interval. In comparison, PRISM increases the batching interval from $\frac{TI}{l}$ to $(TI - l * \epsilon)$ to significantly reduce communication overhead.

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 [48, 50], i.e., updates by a live node will eventually be visible to probes by connected nodes.

Bawa et. al [7] 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* [21] 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 [11, 53] 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. To address this fundamental problem, PRISM uses NI to condition AI and TI guarantees.

Several aggregation systems have worked to address the failure amplification effect. To mask failures, TAG [30] 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. Other studies have proposed multi-path routing methods [11, 21, 27, 31, 33] for fault-tolerant aggregation. In broadcast wireless networks, multi-routing may be relatively inexpensive compared to wired networks where these aggregation topologies incur bandwidth overhead proportional to the number of multiple paths. Furthermore, in both cases, double-counting can occur for duplicate-sensitive aggregates such as SUM. In comparison, PRISM uses redundant aggregation trees for improving availability and NI to quantify consistency of the aggregation result.

Recent proposals [7, 11, 31, 33] 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 [16] 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 [33], 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 the 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.) These 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 4.4.4 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.

Some recent studies [24, 25, 28] have proposed monitoring systems with distributed triggers that fire when an aggregate of remote-site behavior exceeds an a priori global threshold. These systems are based on a single-level tree hierarchy where the central coordinator tracks aggregate time-series data by setting local filters at remote sites. 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] Details omitted for double-blind reviewing, see http://prism2007.googlepages.com for technical report.

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

[3] D. Andersen, H. Balakrishnan, M. Kaashoek, and R. Morris. Resilient overlay networks. In *Proc. SOSP*, pages 131–145. ACM Press, 2001.

[4] T. Anderson and M. Reiter. Geni facility security. In *Draft GDD-6-23, GENI Distributed Services Working Group*, 2006.

[5] T. Anderson and T. Roscoe. Learning from planetlab. In *WORLDS*, 2006.

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

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

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

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

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

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

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

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

[14] A. Deshpande, S. Nath, P. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. In *Proc. SIGMOD*, 2003.

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

[16] 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.

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

[18] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An architecture for secure resource peering. In *Proc. SOSP*, Oct. 2003.

[19] 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.

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

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

[22] 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.

[23] 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.

[24] L. Huang, M. Garofalakis, J. Hellerstein, A. Joseph, and N. Taft. Toward sophisticated detection with distributed triggers. In *MineNet*, pages 311–316, New York, NY, USA, 2006. ACM Press.

[25] L. Huang, M. Garofalakis, A. D. Joseph, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. Technical Report UCB/EECS-2006-139, EECS Department, University of California, Berkeley, October 30 2006.

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

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

[28] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *HotNets*, San Diego, CA, November 2004.

[29] P. Laskowski and J. Chuang. Network monitors and contracting systems: competition and innovation. In *SIGCOMM*, 2006.

[30] 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.

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

[32] 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.

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

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

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

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

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

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

[39] G. Plaxton, R. Rajaram, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *SPAA*, pages 311–320, June 1997.

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

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

[42] 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.

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

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

[45] 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.

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

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

[48] 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.

[49] D. Veitch, S. Babu, and A. Pasztor. Robust synchronization of software clocks across the internet. In *IMC*, 2004.

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

[51] 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.

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

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

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

[55] 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.

[56] 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.