WebView: Scalable Information Monitoring for Data-Intensive Web Applications

Navendu Jain
nav@cs.utexas.edu

Mike Dahlin
dahlin@cs.utexas.edu

Yin Zhang
yzhang@cs.utexas.edu

Department of Computer Sciences
University of Texas at Austin
Austin, TX 78712

Abstract

We present WebView, a scalable information monitoring service for data-intensive Web applications that continuously monitors local application state, aggregates local state into a global view, and uses the global view to help ensure high performance and high availability for these applications. We demonstrate the effectiveness of WebView by building three key Web applications: (a) a data prefetching service for content distribution and (b) a heavy hitter monitoring service for detecting anomalies such as flash crowds and denial-of-service attacks, and (c) monitoring large-scale systems hosting Web services.

To provide a global system view in real-time, a monitoring service faces two key challenges: (1) scalability to many nodes and attributes as well as high volume of data updates and (2) bounding the *freshness* of monitoring results i.e., the time between an event update and its notification to the application. To address these challenges, we design, implement, and evaluate WebView that leverages Distributed Hash Tables (DHT) to build scalable aggregation trees, and that exploits precision-performance tradeoffs which tolerate a bounded small approximation error to significantly reduce the monitoring load. WebView enables applications to control this tradeoff by providing (1) *arithmetic filtering* that caches recent reports and only transmits new information if it differs by some numeric threshold (e.g., $\pm$ 10%) from the cached report and (2) *temporal batching* that combines multiple updates that arrive near one another in time into a single network message, and that further bounds the freshness of monitoring query results. Our prototype implementation of WebView combines these techniques with DHT-based aggregation hierarchy to implement a single highly-scalable monitoring system. Evaluation of our WebView prototype for our three Web applications shows that our system provides significant application benefits and is an order of magnitude more scalable than existing approaches while still delivering fresh results with high accuracy.

1. INTRODUCTION

With the growing scale of the World Wide Web (WWW), an increasingly key requirement is managing high-performance, secure, and high-availability Web services. To meet these demands, we require constant measurement and monitoring of these services by gathering and correlating information from many vantage points in the network. For example, in a cooperative caching system, a set of caches can gather local object access distributions and use these aggregate statistics to prefetch "globally popular" objects for improving response times. Similarly, to balance load on a large set of Web servers or enterprise systems hosting these services requires collecting several attributes from each machine such as request rate, CPU load, memory consumption, etc. Finally, to detect network events such as flash crowds and distributed denial-of-service (DDoS) attacks requires monitoring and analyzing traffic originating from different points in the network.

The common theme binding these applications is a key building block for scalable system monitoring that tracks the distributed application state, performs queries, and reacts quickly to changes in their global state. Ideally, we want to provide this global information view of the system at a minimum cost and in as close to real-time as possible.

To achieve this objective, a monitoring system should address two key challenges. First, the system should scale to many nodes and allow applications to query many data attributes (e.g., CPU load of a Web server, popularity of a cached object). To support large numbers of dynamic attributes, the system must provide high performance to process high volumes of data updates while delivering query results in real-time. Second, the system should deliver time-critical results within a bounded "freshness" delay both for making online decisions (e.g., stock quotes, speed auctions) and detecting anomalies (e.g., DDoS attacks). Unfortunately, traditional monitoring approaches of logging the entire data at a single repository would either generate monitoring traffic whose volume is proportional to the total traffic, or be too late in detecting network anomalies (e.g., DDoS attacks), both of which are clearly unacceptable.

To address these challenges, we have developed WebView, a scalable information monitoring service for data-intensive Web applications that provides a global information view of these applications in real-time. To compute this global view, WebView provides *hierarchical aggregation* [17, 21, 37, 42] that allows a node to access detailed views of nearby information and summary views of global information. To provide aggregation hierarchy, WebView builds on recent work that uses DHTs [28, 30, 31, 34] to construct scalable, load-balanced forests of self-organizing aggregation trees [3, 12, 28, 42]. To process high volume of data updates in real time, WebView provides two key techniques vital for a monitoring system's scalability.

- *Arithmetic filtering* [21, 25] caches recent reports and only transmits new information if it differs by some nu-

WWW2008, April 21–25, 2008, Beijing, China.

meric threshold (e.g., ± 10%) from the cached report, thus trading a bounded small approximation error for a significant load reduction. WebView provides the basic mechanisms to perform arithmetic filtering in an aggregation hierarchy that divide the numeric error among different nodes in the tree while bounding the total error.

- *Temporal batching* [23, 32] combines multiple updates that arrive near one another in time into a single network message. A key benefit of temporal batching is that it bounds the freshness of monitoring query results by taking into account the inherent delay to propagate updates through the system. To implements temporal batching, WebView pipelines the available batching interval across levels of the aggregation hierarchy to maximize the number of updates batched together.

A key focus of this paper is to demonstrate the effectiveness of WebView by building three key Web applications: (1) data prefetching for content distribution, (2) Distributed Heavy Hitter detection (DHH), and (3) PrMon, a monitoring application for large-scale systems hosting Internet-scale services. Our experience with building these applications using WebView illustrate how explicitly managing precision-performance tradeoffs can qualitatively enhance a monitoring service. The key benefit is improved scalability: for these applications, introducing small amounts of arithmetic imprecision (AI) and temporal imprecision (TI) drastically reduces 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 periodic centralized logging. Conversely, PrMon can provide highly responsive monitoring with a TI guarantee of 30 seconds and a 10% AI for approximately the same network cost as once-per-5-minute periodic logging. Another significant advantage of WebView is the benefit to applications: by gathering popularity information from several cooperating caches, our WebView based data prefetching service can significantly increase hit rates compared to demand caching by almost 3x for about 2.5x increase in bandwidth cost.

The key contributions of this paper are as follows[1]. First, we present WebView, the first DHT-based system that enables precision-performance tradeoffs for scalable aggregation and continuous monitoring for large-scale Web applications. Second, we provide key mechanisms and efficient implementations of (1) distributing numeric error budgets in an aggregation tree and (2) pipelining of temporal batching delays across tree levels. Third, our evaluation demonstrates that WebView enables scalable aggregation by significantly reducing load for our case-study Web applications. Finally, our prototype implementation demonstrates that WebView provides significant application benefits.

The rest of this paper is organized as follows. Section 2 provides background description of SDIMS [42], a scalable DHT-based aggregation system at the core of WebView. Section 3 describes the WebView design: (a) mechanism and policies to perform arithmetic filtering and (b) pipelining of updates in temporal batching for reducing monitoring overhead. Section 4 presents the experimental evaluation of WebView. Finally, Section 5 discusses related work, and Section 6 provides conclusions.

[1]Note to reviewers: In prior work, we presented a self-tuning algorithm for setting AI [21]. This paper provides significant extensions not explored in earlier work: (1) setting monitoring budget for AI filtering, (2) design and implementation of TI, (3) combining AI and TI in WebView, and (4) experience with Web applications. Finally, all results in this paper are new.

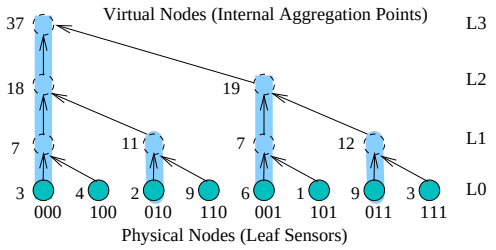

Figure 1: The aggregation tree for key 000 in an eight node system. Also shown are the aggregate values for a simple SUM() aggregation function.

2. POINT OF DEPARTURE

WebView extends SDIMS [42] which embodies two key abstractions for scalable monitoring: aggregation and DHT-based aggregation.

2.1 Aggregation

Aggregation is a fundamental abstraction for scalable monitoring [3, 12, 17, 28, 37, 42] because it allows applications to access summary views of global information and detailed views of rare events and nearby information.

SDIMS'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) [15, 42]. 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.

SDIMS's tree-based aggregation 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, SDIMS 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.

2.2 DHT-Based Aggregation

SDIMS leverages DHTs [28, 30, 31, 34, 45] to construct a forest of aggregation trees and maps different attributes to different trees [3,12,28,42] 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 [3,12,28,42].

2.3 Case-study Applications

Aggregation is a building block for many distributed applications such as network management [43], service placement [13], sensor monitoring and control [23], multicast tree construction [37], and naming and request routing [7]. In this paper, we focus on three case-study applications built using WebView: (1) a data prefetching service for content distribution, (2) a distributed heavy hitter detection service, and (3) a distributed monitoring service for network platforms hosting Web and Internet-scale services.

Data Prefetching for Content Distribution. Our first case-study application is a data prefetching service for Web proxies and content distribution networks (CDN). The aim is to improve hit rates of caches and CDN servers by prefetching and replicating a set of valuable objects that are likely to be referenced in the near future. To identify which objects to prefetch, WebView aggregates local object access patterns from a distributed federation of caches to compute global object popularities. Based on these aggregate statistics, different policies can be applied to select a good set of globally valuable objects for prefetching that will result in significant improvement in hit rates at modest costs.

For our study, we use the Good-Fetch algorithm [39] that balances the object access frequency and object update frequency and that only fetches objects whose probability of being accessed before being updated exceeds a specified threshold. In particular, assuming the overall object access rate to be a, the object access frequency p_i and average lifetime l_i for object i, the probability that i is accessed during its lifetime can be expressed as:

$$P_{goodFetch} = 1 - (1 - p_i)^{al_i} \quad (1)$$

WebView applies this algorithm for a cooperative caching network by aggregating a and the total p_i for all the objects from all caches. Note that our focus is scalably computing these statistics to determine their benefit to applications and not comparing different prefetching algorithms.

Distributed Heavy Hitter detection (DHH). Our second 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 [11, 21]. The key challenge for this distributed query is scalability for aggregating per-flow statistics for tens of thousands to millions of concurrent flows in real-time. For example, a subset of the Abilene [1] traces used in our experiments include 170 thousand flows that send about 50 million updates per hour.

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

PrMon. The final case-study application is PrMon, a distributed monitoring service that is representative of monitoring enterprise data centers hosting Web applications [35] and Internet-scale systems such as PlanetLab [27] and Grid systems [36] that provide platforms for developing, deploying, and hosting global-scale services. For instance, to manage a wide array of user services running on the PlanetLab testbed, the system administrators need a global view of the system to identify problematic services (slices in PlanetLab terminology) e.g., if any slice is consuming more than 10GB of memory across all nodes on which it is running. Similarly, users require system state information to query for "lightly-loaded" nodes for deploying new experiments or to track the resource consumption of their running experiments.

To provide such information in a scalable way and in real-time, WebView 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.

Compared to the common approach of periodically reporting all events to a centralized repository, in Section 4 we show that our system can monitor a larger number of attributes at much finer time scales while incurring significantly lower network costs.

3. WEBVIEW DESIGN

In this section we present the WebView design and describe how to enforce arithmetic and temporal precision limits and quantify the consistency guarantees of query results. WebView's core architecture is a DHT-based aggregation system that achieves scalability by mapping different attributes to different aggregation trees [12, 28, 42]. WebView then introduces controlled tradeoffs between precision guarantees and load.

3.1 Overview

WebView quantifies query precision along two axes: Arithmetic and Temporal. *Arithmetic imprecision* (AI) bounds the numeric difference between a reported value of an attribute and its true value [26,44], 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 [32, 44]. These aspects of precision 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. In the following two subsections, we describe how WebView enforces arithmetic and temporal imprecision bounds while maximizing load reduction.

3.2 Arithmetic Imprecision

Arithmetic imprecision (AI) deterministically bounds the numeric difference between a reported value of an attribute and its true value [20,26,44]. For example, a 10% AI bound ensures that the reported value either underestimates or overestimates the true value by at most 10%.

When applications do not need exact answers and data

values do not fluctuate wildly, AI can greatly reduce monitoring load by allowing caching to filter small changes in aggregated values. Furthermore, for applications like DHH monitoring, AI can completely filter out updates for most "mice" flows i.e., attributes with low frequency.

We first describe the basic mechanism for enforcing AI for each aggregation subtree in the system. Then we describe how our system addresses the policy questions of setting total budgets, computing AI error range, and distributing an AI budget across subtrees to minimize system load.

3.2.1 Mechanism

To enforce AI, each aggregation subtree T for an attribute has an error budget δ_T that 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 since it is likely to be common in network monitoring and financial applications [14]; other standard aggregation functions (e.g., MAX, MIN, AVG, etc.) are similar and defined precisely in an extended technical report [20].

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

$$\begin{array}{rcl} V_{min} & \leq & \sum_{c \in children} V^c_{min} \\ V_{max} & \geq & \sum_{c \in children} V^c_{max} \end{array} \quad (2)$$

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

Note the tradeoff in splitting δ_T between δ_{self} and δ_c. A large δ_c allows a child to filter updates before they reach its parent. 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.

WebView 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. WebView 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 function specifies how δ budgets are distributed (partitioned) among the children and δ_{self}.

3.2.2 Policy Decisions

Given these mechanisms, there is considerable flexibility 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) divide δ_T among δ_{self} and δ_c for each child c.

Setting δ_{root}:. 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, we set δ_{root} for each flow to be at most 1% of the bandwidth of the largest flow measured in the system.) Our mechanisms support 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 $\sum_c V^c_{min}$ 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. 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. For example, we can set the $bias$ ($\in [0, 1]$) parameter as follows:

- $bias \approx 0.5$ if inputs are expected to be stationary
- $bias \approx 0$ if inputs are expected to be increasing
- $bias \approx 1$ if inputs are expected to be 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 it reports [0, 10] to its parent. Then, suppose 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 the node reports [10.5, 20.5] to its parent to filter out small deviations around the current position. Conversely, if $bias = 0$, the node reports [12, 22] to filter out the maximal number of updates of increasing values.

Self-Tuning Error Budgets. A 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 $\delta_{self} = 0.1\delta_T$ 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*, we provide an adaptive algorithm. The high-level idea is simple: increase δ for nodes with high load and large standard deviation but low δ (relative to other nodes); decrease δ for nodes with low load and small standard deviation but high δ.

Additional implementation details of the self-tuning algorithm for distributing error budgets in a general aggregation hierarchy are described elsewhere [21].

3.3 Temporal Imprecision

Temporal imprecision bounds the delay from when an event/update occurs until it is reported. 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. In WebView, each attribute has a TI bound, and to meet this bound, the system must ensure that updates propagate from the leaves to the root in the allotted time.

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

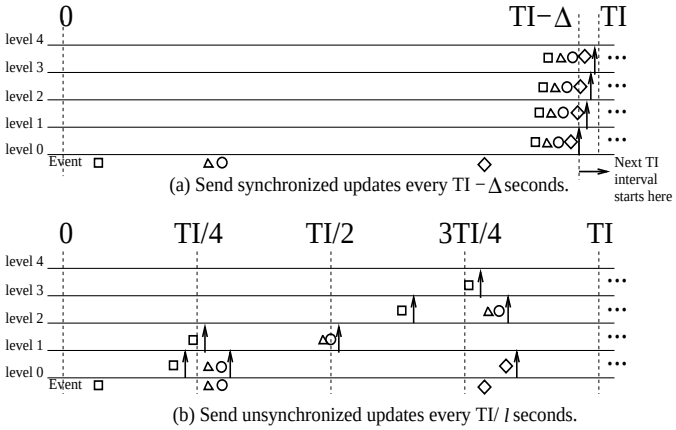

Figure 2: For a given TI bound, pipelined delays with synchronized clocks (a) allows nodes to send less frequently than unpipelined delays without synchronized clocks (b).

tree. Second, explicitly exposing TI allows WebView to use temporal batching: a set of updates at a leaf sensor are condensed into a periodic report or a set of updates that arrive at an internal node over a time interval are combined before being sent further up the tree [20]. This temporal batching improves scalability by reducing processing and network load.

To maximize the possibility of batching updates, when clocks are synchronized[2], we pipeline delays as shown in Figure 2(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 [20] 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 2(b).

3.4 Robustness

Failures and reconfigurations are common in large scale systems. As a result, a query might return a stale answer either due to incorrect previous aggregate values or due to nodes whose inputs are needed to compute the aggregate result becoming unreachable. More importantly, in a large scale monitoring system, such failures interact badly with our techniques for providing scalability—hierarchy, arithmetic filtering, and temporal batching. For example, if a monitoring subtree is silent over an interval, it is difficult to distinguish between two cases: (a) the subtree has sent no updates because the inputs have not significantly changed or (b) the inputs have significantly changed but the subtree is unable to transmit its report. As a result, reported results can be arbitrarily far from their true values.

[2]Algorithms in the literature can achieve clock synchronization among nodes to within one millisecond [38].

Addressing this fundamental problem of node failures and network disruptions in large scale system monitoring is beyond the scope of this paper. In a separate work, we have developed a new metric called Network Imprecision (NI) that characterizes and quantifies the accuracy of query results in the face of failures, network disruptions, and system reconfigurations; the details are available in an extended technical report [20].

4. EXPERIMENTAL EVALUATION

We have developed a prototype of the WebView monitoring system on top of FreePastry [31]. To guide the system development and to drive the performance evaluation, we have also built three case-study applications using WebView as described in Section 2.3.

We performed both micro-benchmark experiments and large scale application experiments to characterize the performance and scalability of the AI and TI metrics. First, we quantify the performance of our system under AI and TI filtering using micro-benchmark experiments. Second, we perform a detailed study analysis of our data prefetching application using Squid traces [33] from IRCache network [18]. For the DHH and the PrMon applications, we use netflow traces from Abilene [1], and CoTop [6] data collected from PlanetLab [27], respectively, to quantify the load reduction due to AI and TI. Third, we analyze the deviation in the WebView's reported values with respect to both the ground truth based on input data readings and the guarantees defined by AI and TI. Finally, we investigate how to set the budgets in terms of AI and TI to reduce the monitoring overhead.

In summary, our experimental results show that WebView is an effective substrate for scalable monitoring: introducing small amounts of AI and TI significantly reduces monitoring load, TI successfully bounds the freshness of reported query results, and Web applications can benefit significantly from the global information view provided by our WebView monitoring system.

4.1 Load vs. Precision

First, we evaluate the performance of WebView using three micro-benchmark experiments.

In the first micro-benchmark, we quantify load reduction for aggregating an attribute solely due to AI with no TI filtering. We compare the monitoring cost of PrMon distributed monitoring service to a centralized periodic logging 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. The CoTop data provides the per-slice resource usage (e.g., CPU, MEM, TX1, etc.) for all applications (slices in PlanetLab terminology) running on a given PlanetLab node. Using these logs as sensor data input, we run WebView on 200 servers mapped to 50 Emulab machines each having a 3GHz CPU and 2GB RAM.

Figure 3 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 (i.e., no AI caching) and TI = TI_{min} = 50ms i.e., immediate update propagation assuming 50ms as the minimum end-to-end propagation delay in the system. Each data point

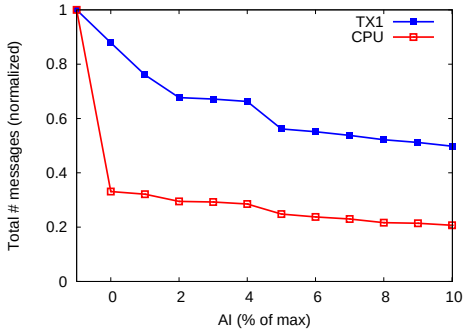

Figure 3: Load vs. AI for TX1 and CPU attributes with no TI filtering.

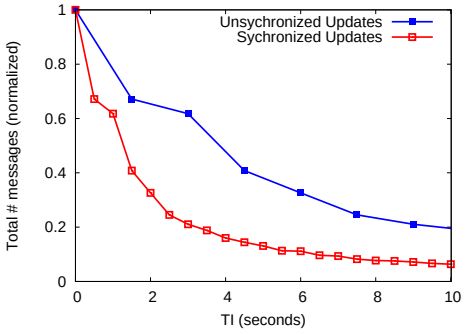

Figure 4: Load vs. TI for a single attribute with no AI filtering.

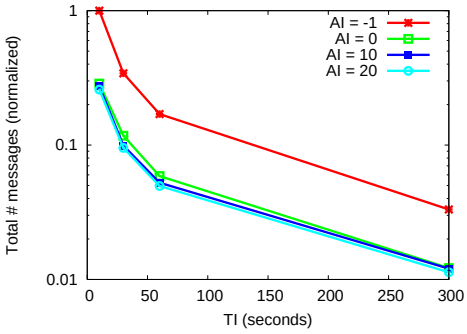

Figure 5: Load vs. AI and TI for PrMon's CPU attribute.

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 even 10% AI. For the TX1 attribute, since the sensor sends an update every 60 seconds, it is relatively stable compared to the dynamic CPU attribute. Hence, it can achieve higher benefits from AI filtering for x-axis ranging from AI = 0 to AI = 10% in Figure 3. In this case, changing AI from -1 to 0 provides roughly a 12% reduction in load whereas 10% AI reduces the load by about 50%.

In the second micro-benchmark, we complement the first experiment by quantifying load reduction for aggregating an attribute solely due to TI with no AI filtering. Figure 4 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 (unsynchronized clocks) and pipelined (synchronized clocks) 10 second TI delays respectively compared to TI of TI_{min}.

The third benchmark evaluates the combined effect of AI and TI in reducing monitoring load. Figure 5 shows the corresponding precision-performance graph 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 4. 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.

4.2 Applications

In this subsection we first perform a detailed study analysis of our data prefetching service for content distribution to evaluate application benefits of WebView, and then quantify the performance of WebView for the DHH and the PrMon applications.

4.2.1 Data Prefetching Service

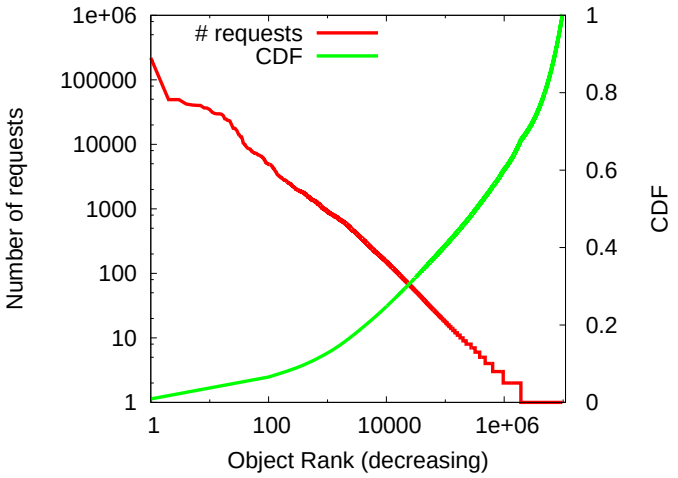

Figure 6: Number of requests and CDF versus object rank for the IRCache dataset. The x-axis and y-axis are on log scale.

We begin by analyzing the hit rates, bandwidth cost obtained by our prefetching service that aggregates popularity information from several cooperating caches. We gathered a 9 day trace logged by the Squid proxy caches [33] operated by the IRCache system [18] between 22 Oct and 30 Oct 2007. The trace consists of 24 million records and accesses to 9.7 million unique objects from 7500 client IPs. The trace also logs the sizes of the objects including the headers information. However, since the traces do not contain object update information, we generated object lifetimes as randomly distributed between 1 and 1,000,000 seconds (11.6 days) given by Wu et al. [41] assuming no correlation with popularity or size. Query URLs (having a "?" in the URL) are considered as both uncacheable and unprefetchable. The Squid proxy mesh implements as a demand cache that routes requests based on network proximity and load balances requests across caches.

Figure 6 shows the number of requests versus object rank and the corresponding cumulative distribution function (CDF) for the IRCache dataset. As expected [4, 5], we observe a Zipf-like popularity distribution of Web objects where the 100 most popular objects account for about 7% of requests, 1000 most popular constitute about 13% of requests, and

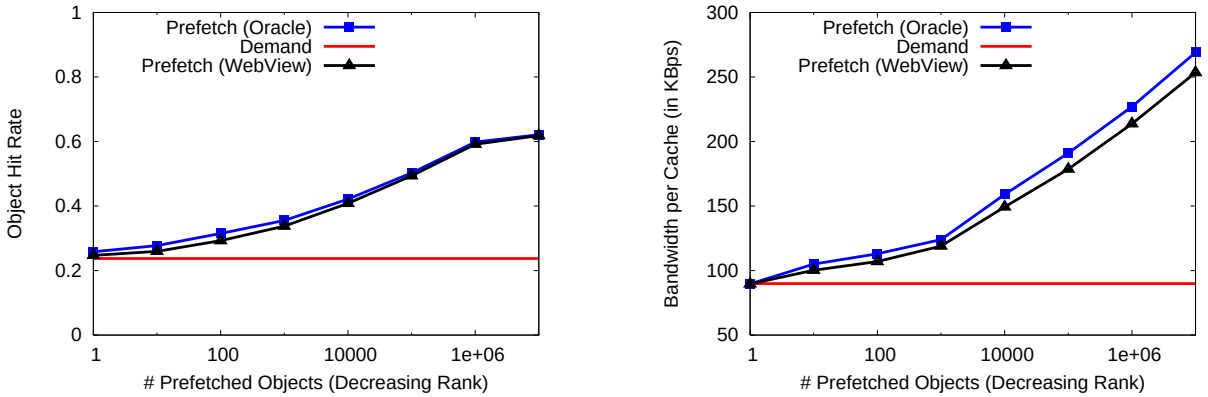

Figure 7: Comparing the performance of WebView compared to oracle and demand caching: (a) Object Hit Rate and (b) Bandwidth usage per cache.

10,000 most popular about 25% of requests. For our evaluation, we uniformally distributed the input trace into 60 buckets corresponding to 60 sensors which are mapped to 30 nodes running on the department Condor cluster each having dual core 2.4GHz and 4GB RAM.

For all experiments in this subsection, we set TI = 10 seconds and AI of 10% of the global access frequency in the most popular object in the trace. Since the input trace data shows a Zipf-like distribution, setting this AI budget likely to filter most of the mice objects i.e., objects that are least popular since they will incur bandwidth cost for prefetching without providing any significant increase in hit rates. However, it may perturb the popularity estimate of medium-popular objects which we next evaluate.

We analyze the performance of WebView using the GoodFetch algorithm [39] compared to pure demand cache and oracle that knows the global access and update pattern of each object apriori. Figure 7(a) plots the hit rates for these three techniques. The x-axis denotes the number of prefetched objects as ranked by the respective prefetching algorithms and y-axis shows the hit rate. Demand caching provides a hit rate of about 22%. As we increase the number of prefetched objects, the hit rates for both WebView and oracle services increases steadily but peaks out after 1 million objects since the latter ranked objects have medium to low popularity and hence least likely to be accessed. WebView computes approximate global statistics due to AI and TI and trails by at most 9% in hit rates from the oracle.

Figure 7(b) shows the corresponding graph for the bandwidth usage per cache. Demand caching incurs a bandwidth cost of about 90 KBps per cache. To compute the bandwidth usage of the oracle, we assume a push-based system in which updates to objects are sent immediately by the server to caches that are "subscribed" to the object in question. Note that as we increase the number of prefetched objects, the bandwidth consumption slowly increases up to 1000 objects and quickly increases after that. From the graph, we observe that oracle consumes roughly 10% more bandwidth compared to our WebView service. Taking the ratio of hit rate to bandwidth consumption in Figure 7, our WebView based data prefetching service can significantly increase hit rates compared to demand caching by almost 3x for about 2.5x increase in bandwidth cost.

4.2.2 Detecting Heavy Hitters

For DHH application, we use multiple netflow traces obtained from the Abilene [1] 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 2.3, 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 8 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.

4.2.3 PrMon

Finally, we evaluate the monitoring cost of PrMon distributed monitoring service using CoTop [6] 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 WebView on 200 servers mapped to 50 Emulab machines. Note that for comparison with centralized services that performs periodic logging, the baseline is set to AI of -1 (no AI caching) and TI of 5 minutes.

Figure 9 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

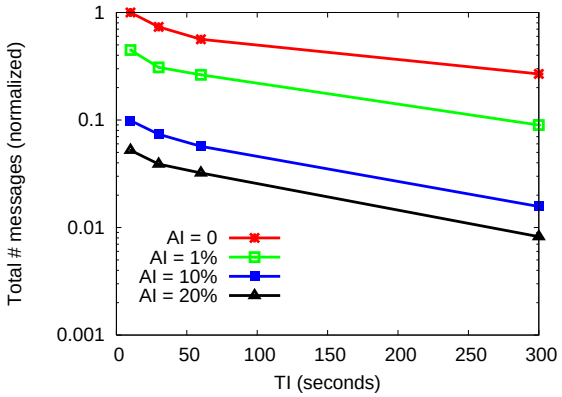

Figure 8: Load vs. AI and TI for DHH application.

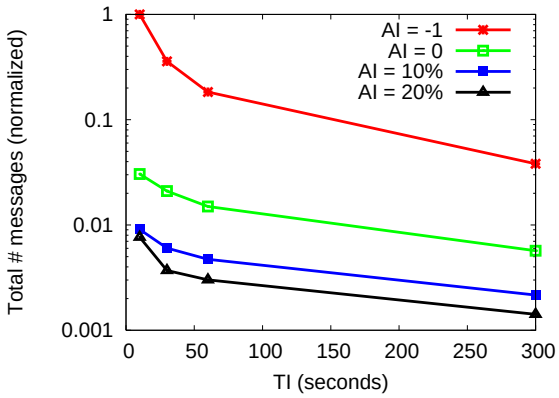

Figure 9: Load vs. AI and TI for PrMon application.

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 centralized logging that sets AI of -1 and TI of 5 minutes. Alternatively, for approximately the same bandwidth cost as periodic logging with TI of 5 minutes and AI of -1 for 200 nodes, WebView provides highly time-responsive and accurate monitoring with TI of 10 seconds and AI of 0.

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

4.3 Setting Monitoring Budget

Finally, to reduce bandwidth, one can either increase AI or TI. We provide two guidelines here. First, TI filtering is more beneficial than AI for attributes that exhibit (a) large variation in values of consecutive updates (e.g., CPU) or (b) high update rates. For these attributes, bandwidth falls roughly proportionally with increasing TI but increasing AI may only have a little impact as most updates will bypass AI filters under modest AI error budgets as shown in Figure 5. Second, AI filtering is more useful for attributes that show small variance in values of consecutive updates (e.g., TX1 attributed in Figure 3, number of processes). Thus, for environments where sensor readings change slowly (or even predictably) with time, increasing the AI budget may be more effective.

4.4 Promised vs. Realized Accuracy

A central goal of WebView is to go beyond providing best effort imprecision estimates to ensuring worst-case approximation guarantees. In this subsection, we experimentally investigate WebView's accuracy by using the CoTop trace for the "CPU" attribute, configuring WebView with different AI and TI values, playing that trace through WebView 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 WebView's reported values and trace inputs.

First, we experimentally test whether the results delivered by WebView do, in fact, remain within the range promised by WebView's imprecision guarantees. We compare WebView'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 WebView; the inaccuracy of less than 1% in the remaining 0.1% of reports stems from short-term disruptions in the system such as heavy CPU load and unexpected network delays.

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 10 and 11 show for different values of AI, the CDF of deviation between WebView's reports compared to the oracle truth for fixed TI of 1s and 10s, respectively. 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 11, increasing the TI to 10 seconds results in a larger deviation between WebView'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.

5. RELATED WORK

Aggregation systems commonly use some form of AI or TI to reduce monitoring overheads. Olston et al. [2, 25] use adaptive filters at the data sources that compute answers with bounded numerical approximation for continuous queries in single-level communications topologies. Manjhi et al. [24] 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. A-GAP provides an adaptive protocol to bound the average approximation error of continuous queries in a spanning tree [29]. In comparison, WebView supports general aggregation functions and employs a self-tuning algorithm for distributing the error budgets in a general communication hierarchy. Further, while many existing approaches [9, 25] periodically shrink error budgets for each aggregation tree which limits scalability for tracking many attributes, WebView's self-tuning algorithm performs is optimized to scale

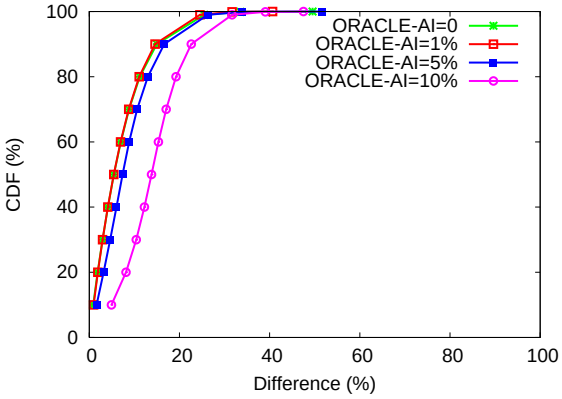

Figure 10: CDF for difference between WebView's reported values and oracle truth for fixed TI of 1 second.

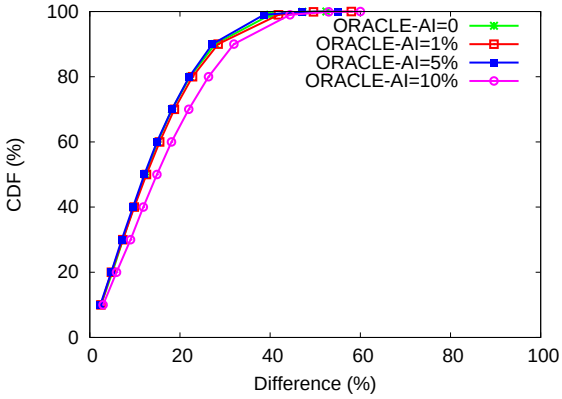

Figure 11: CDF for difference between WebView's reported values and oracle truth for fixed TI of 10 seconds.

to a large number of nodes and attributes [21].

IrisNet [10] 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, WebView coordinates transmission of push-based continuous query results to support in-network aggregation, allowing it to more aggressively optimize the TI batching. TAG [23] 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, WebView 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 [37, 42], 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 Web, P2P, and databases communities to build global monitoring services. PIER is a DHT-based relational query engine [17] targeted at querying real-time data from many vantage-points on the Internet. Sophia [40] is a distributed monitoring system designed with a declarative logic programming model. ATMEN [22] is a distributed triggered measurement infrastructure which uses Gigascope [8] packet monitoring system. ATMEN focuses on monitoring the performance of Web sites and DNS servers from multiple vantage points while WebView's goal is to scalably aggregate local information into a global view of the system. Gupta et al. [14] propose a query planning based approach to answer continuous queries in dynamic data dissemination networks like CDNs.

Some recent studies [16,19] 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. WebView may enhance such efforts by providing a scalable way to track top-k and other significant events.

6. CONCLUSIONS

We designed, implemented, and evaluated WebView—a scalable information monitoring infrastructure for performing data aggregation and continuous event monitoring for data-intensive Web applications. To the best of our knowledge, WebView is the first DHT-based system that enables applications to control precision-performance tradeoffs by specifying the precision guarantees of their query results and then have the monitoring system minimize the load under those precision constraints. To support these tradeoffs, WebView provides key mechanisms and efficient implementations of (1) distributing AI error budgets in an aggregation tree and (2) pipelining TI delays across tree levels. Our evaluation demonstrates that WebView provides significant application benefits and enables scalable aggregation by significantly reducing load for three key Web applications we studied.

In future work, we plan to examine techniques for secure information aggregation in distributed systems spanning multiple administrative domains as well as develop a broad range of applications that could benefit from WebView.

7. REFERENCES

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

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD*, June 2003.

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

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *INFOCOM (1)*, pages 126–134, 1999.

[5] J. Cho. Crawling the web: Discovery and maintenance of large-scale web data, 2001.

[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] C. D. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *SIGMOD*, 2003.

[9] A. Deligiannakis, Y. Kotidis, and N. Roussopoulos. Hierarchical in-network data aggregation with quality guarantees. In *EDBT*, 2004.

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

[11] C. Estan and G. Varghese. New directions in traffic

measurement and accounting. In *SIGCOMM*, 2002.

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

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

[14] R. Gupta and K. Ramamritham. Optimized query planning of continuous aggregation queries in dynamic data dissemination networks. In *WWW*, pages 321–330, 2007.

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

[16] L. Huang, M. Garofalakis, A. D. Joseph, and N. Taft. Communication-efficient tracking of distributed cumulative triggers. In *ICDCS*, 2007.

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

[18] `http://www.ircache.net` .

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

[20] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. PRISM: Precision-Integrated Scalable Monitoring (extended). Technical Report TR-06-22, UT Austin Department of Computer Sciences, 2006.

[21] N. Jain, D. Kit, P. Mahajan, P. Yalagandula, M. Dahlin, and Y. Zhang. Star: Self tuning aggregation for scalable monitoring. In *VLDB*, 2007.

[22] B. Krishnamurthy, H. V. Madhyastha, and O. Spatscheck. Atmen: a triggered network measurement infrastructure. In *WWW*, 2005.

[23] 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.

[24] A. Manjhi, V. Shkapenyuk, K. Dhamdhere, and C. Olston. Finding (Recently) Frequent Items in Distributed Data Streams. In *ICDE*, 2005.

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

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

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

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

[29] A. G. Prieto and R. Stadler. Adaptive distributed monitoring with accuracy objectives. In *INM*, 2006.

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

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

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

[33] `http://www.squid-cache.org`.

[34] 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.

[35] C. Tang, M. Steinder, M. Spreitzer, and G. Pacifici. A scalable application placement controller for enterprise data centers. In *WWW*, 2007.

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

[37] 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.

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

[39] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. In *WCW*, 2001.

[40] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *HotNets-II*, 2003.

[41] B. Wu. Objective-optimal algorithms for long-term web prefetching. *IEEE Trans. Comput.*, 55(1), 2006. Senior Member-Ajay D. Kshemkalyani.

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

[43] P. Yalagandula, P. Sharma, S. Banerjee, S.-J. Lee, and S. Basu. S^3: A Scalable Sensing Service for Monitoring Large Networked Systems. In *INM*, 2006.

[44] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *TOCS*, 2002.

[45] 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.