Self-Configuring Information Management for Large-Scale Service Overlays

Jin Liang* , Xiaohui Gu† and Klara Nahrstedt*

*Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801
{jinliang, klara}@cs.uiuc.edu
†Department of Distributed Computing
IBM T.J. Watson Research Center, Hawthorne, NY 10532
xiaohui@us.ibm.com

Abstract— **Service overlay networks (SON) provide important infrastructure support for many emerging distributed applications such as web service composition, distributed stream processing, and workflow management. Quality-sensitive distributed applications such as multimedia services and on-line data analysis often desire the SON to provide up-to-date dynamic information about different overlay nodes and overlay links. However, it is a challenging task to provide scalable and efficient information management for large-scale SONs, where both system conditions and application requirements can change over time. In this paper, we present InfoEye, a model-based self-configuring distributed information management system that consists of a set of monitoring sensors deployed on different overlay nodes. InfoEye can dynamically configure the operations of different sensors based on current statistical application query patterns and system attribute distributions. Thus, InfoEye can greatly improve the scalability of SON by answering information queries with minimum monitoring overhead. We have implemented a prototype of InfoEye and evaluated its performance using both extensive simulations and micro-benchmark experiments on PlanetLab. The experimental results show that InfoEye can significantly reduce the information management overhead compared with existing approaches. In addition, InfoEye can quickly reconfigure itself in response to application requirement and system information pattern changes.**

I. INTRODUCTION

Federated computing infrastructures such as computational grids [7] and service overlay networks (SON) have become increasingly important to many emerging applications such as web service composition, distributed stream processing, and workflow management. As these computing infrastructures continue to grow, how to efficiently manage such large-scale dynamic distributed systems to better support application needs has become a challenging problem. Distributed information management service[13], [15], [10] is one of the fundamental building blocks of system management, which can track dynamic system information and make it available via some query interfaces. Applications running in the distributed environment can then query the current status of the system and make appropriate management decisions.

Figure 1 shows a typical federated distributed system consisting of (1) overlay nodes that execute different application tasks; (2) management nodes that monitor the status of all
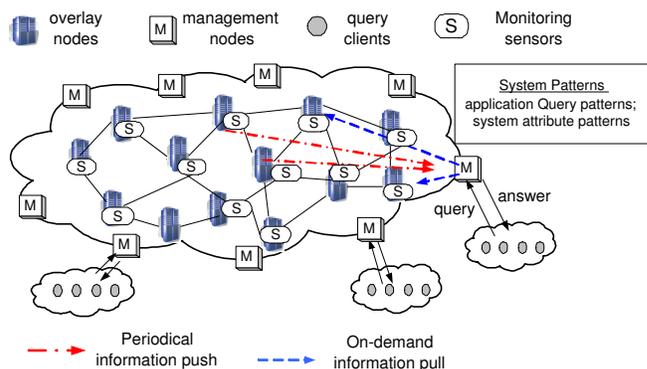

Fig. 1. Pattern-driven distributed information management systems.

overlay nodes and perform system management tasks (e.g., job scheduling, resource allocation, system trouble-shooting); (3) *monitoring sensors* that monitors and provides information of each host to management nodes. The information management system resides within the management nodes, which can resolve information queries from other system management modules or user applications.

However, providing scalable and efficient information management service for *large-scale, dynamic* distributed systems such as SONs is a challenging task. On one hand, quality sensitive applications running in such environments desire up-to-date information about the current system in order to better accomplish their application goals. On the other hand, the system can include a large number of geographically dispersed nodes (e.g., the World Community Grid [2] consists of many thousands of nodes), and each node can be associated with many dynamic attributes (e.g., CPU load, memory space, disk storage, and other application level attributes). Obtaining accurate information about all nodes with their complete information would inevitably involve high system overhead.

Previous work [13], [10], [15] has investigated different architectures for scalable information management. In this paper, we explore a new design dimension, i.e., utilizing the statistical patterns of application needs and system conditions to intelligently configure the information management system for minimum monitoring overhead. As a simple example, if

most of the application queries require at least 20% available CPU time, then it is unnecessary to monitor a node that has only 10% CPU time, because it is unlikely to match any queries. Although the idea is simple, the challenging question is *what* statistical patterns can be utilized, and *how*?

This paper presents InfoEye, a novel pattern-driven, self-configuring distributed information management system that provides answer to the above questions. Briefly, InfoEye is based on the observation that for any information management system, there are essentially two approaches to dynamic information collection: (1) *information push* where overlay nodes periodically report their current attribute data to the management nodes; and (2) *information pull* where the management nodes dynamically request information from overlay nodes for query resolution.

Each of the approach has its merits. For example, push is more efficient when the query arrival rate is high, because the push cost is amortized among many queries; and pull is more efficient when the query arrival rate is low, because it only collects data that are needed. Hence, in a dynamic distributed system where both query patterns and system conditions can change over time, any static solutions (i.e., statically configured push or pull operations) are insufficient. To achieve scalability and efficiency, a distributed information management system must be able to adaptively configure itself based on current query patterns and system conditions.

In order to achieve such adaptivity, we develop an analytical model that precisely captures the relationship between system cost and various system parameters. This model allows InfoEye to dynamically configure the information management parameters in order to utilize various application and system patterns such as query arrival rate, attribute popularity, and system resource distribution.

We have implemented InfoEye and conducted both simulation studies and micro-benchmark experiments on the PlanetLab[1]. Our results show that by exploiting application query patterns and statistical system conditions, InfoEye can achieve much lower management overhead than static approaches that are agnostic to these patterns. In addition, when the query patterns or system conditions change, InfoEye can always re-configure itself to the best operating point.

The rest of the paper is organized as follows. Section II gives an overview of the InfoEye system, including the system model and problem description. Section III presents the the detailed configuration algorithm for efficient information management. Section IV presents the experimental evaluation results. Section V discusses related research work. Finally, Section VI concludes the paper.

II. INFOEYE OVERVIEW

In this section, we present a high level overview of the InfoEye system. The notations used in this and subsequent

---

sections are summarized in Table I.

A. System Model

We consider a distributed system that has N overlay nodes to be monitored, as is shown in figure1. Each node is associated with a set of attributes (e.g., CPU load, number of disk accesses) that are denoted by $A = \{a_1, ..., a_{|A|}\}$. Each attribute a_i is denoted by a name (e.g., CPU, memory) and value (e.g., 10%, 20KB)[2].

A management node is responsible for monitoring the distributed system. It provide information to the application by answering their information queries. Although in a real system there can be multiple management nodes, in this paper we focus on exploiting statistical application patterns and consider the algorithm in a single management node. To extend the idea to multiple management nodes, the management nodes need to share statistical information among themselves. However, further discussion of this is outside the scope of this paper.

For applications such as service composition and distributed stream processing, the query can often be expressed as locating some overlay nodes that have certain resources, e.g., $(a_1 \in [l_1, h_1]) \wedge (a_2 \in [l_2, h_2])... \wedge \cdots (a_k \in [l_k, h_k])$, where l_i and h_i are the desired lower bound and upper bound for a_i, respectively. Each query can also specify the number of overlay nodes that are needed. The query answer should return the specified number of overlay nodes, each of which satisfies the query predicate. Finally, each query can also specify a staleness constraint T_i on a required attribute a_i, which means the attribute value used to resolve this query should be no more than T_i seconds old. The staleness constraint is meant to give applications more specific control on their query result. If a query does not specify such constraint, a default value (e.g., 30 seconds) can be used instead.

On each overlay node, there is a monitoring software called a monitoring sensor. The monitoring sensor can be configured by the management node to periodically push its information only when certain conditions are satisfied. It can also respond to a dynamic probe with its current information. Such configurability allows the management node to achieve adaptiveness based on statistical query patterns.

B. Statistical Patterns

InfoEye performs automatic self-configuration based on dynamically maintained statistical information about the queries and system conditions. Specifically, the current InfoEye prototype maintains the following statistical information:

Frequently queried attributes. Although overlay nodes can be associated with many attributes, it is likely only a subset of them are frequently queried by current applications. For example, in distributed applications where computing jobs are mainly CPU-bound, most queries will specify requirements on the CPU resource, but not on other attributes. As a result, the management node can configure the monitoring sensors to only push the subset of attributes (denoted as A^*) that

---

| notation	meaning	notation	meaning
N	total number of overlay nodes	a_i	system state attribute
A	set of all attributes	A^*	subset of attributes to be pushed
$f_1 = \frac{|A^*|}{|A|}$	fraction of pushed attributes	T	push interval
T_i^*	optimal push interval for a_i	T_i	staleness constraint of a query
S_1	size of push message	S_2	size of probe message
λ	average query arrival rate	n	average probing overhead
p_1	% of resolvable queries using A^*	l_i	lower bound requirement for a_i
l_i^*	(optimal) push threshold for a_i	f_2	% nodes in the push subspace
p_2	% queries in the push subspace	p_3	% queries satisfied by the push intervals

TABLE I

NOTATIONS.
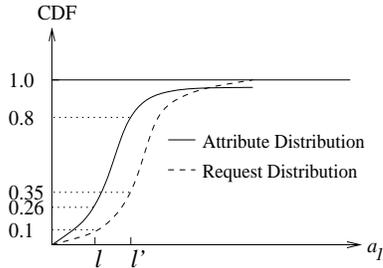
Fig. 2. System cost under different configurations.

are likely to be queried. This allows the management node to resolve queries that only specify attributes in A^*. For other queries, dynamic probe (pull) can be invoked for their resolution.

Frequently queried range values. Besides selecting popular attributes, we can further reduce the system cost by filtering out unqualified attribute values. For example, if most queries on CPU time require a node to have at least 20% free CPU time, the nodes with less than 10% CPU free time do not need to push their CPU value since they are unlikely to satisfy the query predicate. Generally, we can configure the monitoring sensor with a push triggering range[3] $[l_i^*, \infty)$ for each selected attribute $a_i \in A^*$. The monitoring sensor will push the attribute data only if the attribute value falls into this range. The lower bound l_i^* of the configured range is called the *push threshold* for the attribute. By properly setting the push threshold, we can filter out a lot of unnecessary data push without significantly decreasing the query hit ratio (i.e., percentage of queries that can be resolved by the pushed data).

Figure 2 illustrates the problem of push threshold selection for one attribute. The solid line is the cumulative distribution function (CDF) of an attribute a_1 across all N nodes, and the dashed line is the CDF of the lower bound requirements from the current queries. As the figure shows, 90% queries require the attribute to be greater than l, and only 74% of nodes satisfy this requirement. If we configure the push threshold to be l, 74% of nodes will push their attribute data and 90% of queries

[3]The query predicates such as in resource queries often do not have upper-bound constraints. Our scheme can also be easily extended to include a finite upper-bound.

can be resolved by the pushed data. However, if we increase the push threshold from l to l', only 20% of nodes need to push their attribute data with a moderate decrease of query hit ratio (from 90% to 65%).

Frequent staleness constraints. The last query pattern that InfoEye utilizes is called frequent staleness constraints. When an application makes a query, it can specify a *staleness constraint* T_i, which means the attribute data used to resolve the query should be no more than T_i seconds old for attribute a_i. It is likely for any attribute $a_i \in A^*$, different queries may have different staleness requirements. As a result, the push interval (i.e., update period) of a_i should be dynamically configured, so that the push frequency is just enough to satisfy the staleness constraints of most queries.

Node attribute distributions. In addition to the query patterns, InfoEye also maintains an estimate of node attribute distribution. The distribution can be used for two purposes. First, we can estimate the probing cost (i.e., the number of probes that will be generated) based on the node attribute distributions. Second, the attribute distributions allow us to estimate the push cost reduction and pull cost increase when we configure the push thresholds for different attributes (in Section III-B). Since our system involves multiple attributes, we maintain multi-dimensional histograms to estimate the attribute distribution, which can be obtained by executing infrequent aggregate queries (e.g., `histogram`) over all the nodes [9].

C. Problem Formulations

Since InfoEye combines the push and pull for data collection, its management cost (or total system cost) includes two parts, *push cost* and *pull cost*. The push cost is the amount of data periodically delivered from different overlay nodes to the management node. The pull cost is the amount of data generated per time unit for pulling the attribute data, in response to queries that cannot be resolved by the management node locally. The goal of InfoEye is to dynamically configure the monitoring sensors, so that the total system cost is minimized.

Corresponding to the application query patterns, there are three configuration parameters that InfoEye can tune. The first is the subset A^* of attributes that are pushed. This means each

monitoring sensor only periodically pushes a subset A^* of attributes. When a query arrives, if all the attributes it specifies is in A^*, no additional cost is incurred. Otherwise, some on-demand probing protocol is needed to find enough nodes that satisfy the query[4].

Since each monitoring sensor periodically (every T seconds) pushes $f_1 = \frac{|A^*|}{|A|}$ percentage of the attributes, assume the message size is proportional to the number of attributes pushed, and S_1 is the size of the message if all $|A|$ attributes are pushed, the push cost of the system can be expressed as by $\frac{1}{T} N f_1 S_1$. Suppose the average query arrival rate is λ and on average we need to probe n nodes with $2n$ messages (probes and replies) to resolve a query by pull. Let p_1 denote the query hit ratio, and S_2 denote the size of a probe message [5], the pull cost of the whole system is $2n(1-p_1)\lambda S_2$. As a result, if only popular attributes are configured, and A^* is the set of selected attributes, the total system cost is

$$\frac{1}{T} N f_1 S_1 + 2n(1-p_1)\lambda S_2. \tag{1}$$

Larger A^* implies larger push cost (i.e., higher f_1) but lower pull cost (i.e., lower $1-p_1$). Thus, InfoEye dynamically selects A^* based on the dynamically maintained statistical information, so that the overall system cost in Equation 1 is minimized.

Given a subset A^* that has been selected, we can further reduce the system cost by selecting a push threshold l_i^* for each attribute $a_i \in A^*$, and filtering out the nodes that do not satisfy the push thresholds. The set of push thresholds define a subspace $\{(a_1, a_2, \cdots, a_{|A^*|}) | a_i \geq l_i^*, 1 \leq i \leq |A^*|\}$ in the $|A^*|$-dimensional space. We say a node is "covered" by the subspace, if its value for each attribute $a_i \in A^*$ is above the push threshold. We say a query is "covered" by the subspace, if its lower bound requirement on each $a_i \in A^*$ is above the push threshold. If a query is covered by the subspace, it means all the nodes that satisfy the query (called the answer set of the query) are covered by the subspace, thus it can be locally resolved safely. For a query not covered by the subspace, its answer set is not completely available. In this case, we assume a probe is invoked, so that the query result is not biased toward a subset of the answer set.

Suppose a overlay node reports its attribute data A^* only if the node is covered by the subspace, and f_2 percent of the overlay nodes are covered by the subspace defined by the push thresholds. The push cost of the system is reduced to $\frac{1}{T} f_2 N f_1 S_1$ since only the f_2 percentage of nodes do periodic push. Correspondingly, if p_2 percent of the queries (among those that only specify attributes in A^*) are covered by the

Fig. 3. Push attribute selection algorithm.

subspace, a total of $(1 - p_1 p_2)$ percent queries need to be resolved by dynamic pull. As a result, the total system cost becomes

$$\frac{1}{T} f_2 N f_1 S_1 + 2n(1-p_2 p_1)\lambda S_2. \tag{2}$$

Lower $l_i^*, 1 \leq i \leq |A^*|$ implies larger push cost (i.e., higher f_2) but lower pull cost (i.e., lower $1-p_2 p_1$). Thus, the goal of InfoEye is to select a set of proper push thresholds l_i^* for all attributes $a_i \in A^*$, such that the total system cost in Equation 2 is minimized.

To further reduce the system cost, each overlay node can push the value of $a_i \in A^*$ every T_i^* seconds when the value is above the push threshold. The push cost for attribute a_i becomes $\frac{1}{T_i^*} f_2 N \frac{S_1}{|A|}$. Thus, the total push cost for all selected attributes is $\sum_{a_i \in A^*} \frac{1}{T_i^*} f_2 N \frac{S_1}{|A|}$. Suppose under the above configuration, p_3 percent of queries (among the $p_2 p_1$ percent of queries that specify attributes in A^* and are covered by the subspace defined by the push thresholds) can satisfy their staleness constraints. Then a total of $(1-p_3 p_2 p_1)$ percent queries need to invoke pull operations. Thus, finally the total system cost for all three configuration parameters is

$$\sum_{a_i \in A^*} \left(\frac{1}{T_i^*} f_2 N \frac{S_1}{|A|}\right) + 2n(1-p_3 p_2 p_1)\lambda S_2. \tag{3}$$

Lower T_i^* means higher push cost but lower pull cost due to higher p_3. Thus, InfoEye needs to properly configure a push interval T_i^* for each attribute $a_i \in A^*$ based on the current queries' staleness requirements, such that total system cost in Equation 3 is minimized. Thus, the problem addressed by the InfoEye system is to properly configure A^*, l^* for each attribute $\in A^*$ and T^* on each sensor to minimize the above system cost.

III. DESIGN AND ALGORITHMS

In this section, we describe our algorithms to achieve optimal information monitoring based on the formula derived in the previous section. Our goal is to minimize the total system cost in Equation (3). For simplicity, we describe the

---

[4] There are different ways for dynamic probing, e.g., using random sampling or on-demand spanning trees [9]. Regardless the particular probing protocol, we assume in order to resolve a query by probing, on average n nodes need to be contacted with $2n$ messages. In practice, n can be obtained from previous probes.

[5] Since it is unlikely for a query to specify requirements on many attributes [3], we assume the message size for both probe and reply is S_2, which is a constant much smaller than S_1. However, this is only for notational simplicity and is not essential to our model.

algorithm in several steps as follows. In practice, the steps are executed in an iterative fashion.

A. Push Attribute Selection

The goal of push attribute selection is to select a subset of attributes $A^* \subseteq A$, so that the total system cost is minimized. According to Equation (1), A^* can affect the push cost (i.e., $f_1 = A^*/A$ percent of complete attribute push cost) and the percentage p_1 of queries that can be resolved by the management node locally. Larger A^* implies a larger push cost but also a larger query hit ratio, while smaller A^* implies smaller push cost but also lower query hit ratio and thus higher pull cost.

Our push attribute selection algorithm is shown in Figure 3. In the figure, C is the collection of attribute subsets, each corresponding to a set of queries (e.g., $A_1 = \{a_1, a_3\}$ corresponds to all queries that specify requirement on a_1 and a_3). $freq(A_i)$ is called the "query frequency" for A_i, which means the percentage of all queries that are represented by A_i. $freq'(A_i) = \sum_{A_j \subseteq A_i} freq(A_j)$ is called the "cumulative query frequency", which means the percentage of queries that can be resolved by the push data if all attributes in A_i are pushed. Given an A_i, if we push all attributes in A_i, we will increase the push cost by $\frac{1}{T} N \frac{|A_i|}{|A|} S_1$, but we also reduce the pull cost by $2n \cdot freq'(A_i) \lambda S_2$, because $freq'(A_i)$ percentage of queries can now be locally resolved. The decrease in pull cost minus increase in push cost is called the "cost reduction", which indicates how the system cost will change if A_i is pushed.

Initially, we set A^* to be empty, which means no attribute is pushed. Thereafter, we repeatedly select the subset A_i with the largest cost reduction, and add A_i to A^*. This is repeated until either all attributes have been added to A^*, or the cost reduction for any remaining attribute subset is negative. Note when A_i is added to A^*, Its attributes should be removed from all other subsets in C. This may create duplicate subsets in C. For example, after the attributes in $A_i = \{a_1, a_2\}$ are removed, the two remaining subsets $\{a_1, a_3\}$ and $\{a_2, a_3\}$ will be the same as each other. These subsets are then merged, and the cumulative query frequency recomputed.

B. Push Threshold Configuration

Given the subset A^* as selected by the push attribute selection algorithm, the push threshold configuration algorithm should select a push threshold l_i^* for each attribute $a_i \in A^*$, so that the total cost as in Equation (2) is minimized.

The idea behind push threshold configuration is similar to push attribute selection. For each attribute $a_i \in A^*$, we normalize the possible value range to $[0, 1.0]$, and divide the range into steps of size d. Initially all the push thresholds are set to 0, which means every node will push its attributes in A^*. At each step, an attribute a_i is selected, and its push threshold increased from l_i^* to $l_i^* + d$. Such an increase will reduce the push cost since fewer overlay nodes are covered by the subspace. However, it also increases the pull cost since more queries are uncovered by the subspace. Thus, at each

---

```
PushThresholdSelection(T, N, A, S₁, S₂, n, λ, A*)
 1. let lᵢ* = 0, for 1 ≤ i ≤ |A*| and f₂ = p₂ = 1
 2. compute min_cost according to Equation(2)
 3. let B and B' be the histogram bins for nodes and queries
 4. while B ≠ ∅ do
 5.     select aᵢ that has the largest cost reduction.
 6.     if the cost reduction is < 0 then break
 7.     increase lᵢ* to lᵢ* + d
 8.     remove all nodes and queries not covered by {lᵢ*}
 9.     subtract the cost reduction from min_cost
10. return {lᵢ*}
```

Fig. 4. Push Threshold Selection Algorithm.

step the attribute a_i is selected in a way that maximizes the net cost reduction. The above process is repeated until either every push threshold has reached its maximum, or there is no attribute with positive cost reduction.

The pseudo code for the push threshold selection algorithm is show in Figure 4. The main difference from push threshold selection is how to compute cost reduction given a particular configuration. In the algorithm, B and B' are the histogram bins for the node attribute and query distribution. Each bin in B or B' is described by a tuple of $|A^*| + 1$ fields. The first $|A^*|$ fields define the bin, and the last field is the percentage of nodes/queries in the bin. For example, $b = (v_1, v_2, \cdots, v_{|A^*|}, 0.1) \in B$ means 10% of the machines have attribute $a_i \in [v_i, v_i + d), 1 \le i \le |A^*|$.

C. Push Interval Selection

Given the selected push attributes A^* and push thresholds $\{l_i^* | a_i \in A^*\}$, The goal of push interval selection is to select push interval T_i^* for each attribute $a_i \in A^*$, so that the total system cost according to Equation (3) is minimized.

The push interval selection algorithm works much the same way as the previous two algorithms and thus it is only briefly described here. Initially the push interval T_i^* for each $a_i \in A^*$ is set to a minimum value (i.e., this is the smallest interval that monitoring sensors can push attribute data periodically). Thereafter, at each step, an attribute is selected and the corresponding push interval incremented (by some constant step size). The attribute is selected so that the increase of its push interval results in the largest cost reduction. This is repeated until every push interval has reached some maximum value, or the increase of any push interval would result in negative cost reduction. The cost reduction is computed as the reduced push cost due to slower push minus the increased pull cost due to more queries being pulled (because their staleness constraint cannot be satisfied by the pushed data).

D. Practical Issues

There are several practical issues that need to be mentioned about our algorithms. First, when we resolve a query by pull, the pulled data can be cached for future query resolution. However, this is unlikely to have a big impact on our query resolution, since the data are not periodically refreshed, thus will timeout within a short period of time. Second, the push

interval selection assumes each attribute is independently pushed. This may be undesirable due to a lot of small messages. This can be solved as follows. Suppose the set of push intervals have been selected, and the smallest push interval is T_i^*, we can normalize every T_j^* to $T_i^* \lfloor \frac{T_j^*}{T_i^*} \rfloor$, which is the largest multiple of T_i^* that is still $\leq T_j^*$. This way other attributes can be piggybacked to the push messages for a_i.

IV. EXPERIMENTAL EVALUATION

In this section we present an experimental evaluation of InfoEye system. We first describe our simulation methodology and results, then present the prototype implementation of InfoEye and our experiment results from the PlanetLab [11].

A. Evaluation Methodology

Our simulator consists of a *query generator* that can generate a range of different kinds of query workload, a *query collection* that captures the statistical query patterns, and three configuration modules (i.e., popular attribute selection, push threshold configuration, and push interval configuration). Unless otherwise specified, the system size is $N = 3000$, the default push interval is $T = 30$ seconds, the total number of attributes is $|A| = 50$, the number of nodes to be probed for each pull is $n = 50$, the push packet size is $S_1 = 1000$ bytes and the probe packet size is $S_2 = 100$ bytes. Our parameters are chosen to represent a "typical" system. For example, in the CoMon [1] monitoring service currently running on the PlanetLab, each resource report contains more than 40 attributes, and has about 900 bytes [6].

Our query generator uses similar methods as previous work [10] for query generation. For each query, we first decide the number of attributes in a query, which is uniformly distributed between $[1, k], 1 \leq k \leq |A|$. Next, the specified number of attributes are selected from A. The probability that an attribute is selected follows the Zipf [4] distribution. After that, the lower bound on each attribute is generated. We assume that the value range of each attribute is divided into 50 equal sized bins (intervals). The lower bound for an attribute is generated according to a Zipf distribution, but biased toward the highest value. To generate node attribute values, we use a probability distribution that mimics the actual attribute distribution we observed on the PlanetLab, namely, most nodes have moderate attribute values, but a small number of nodes will have very large or very small attribute values.

We use the total system cost defined in Section II-C as the main evaluation metric. For each experiment, we first generate a set of "training queries" (usually 2000 of them) using the query generator. The query arrival follows a Poisson process with a mean arrival rate λ. We then run our algorithms to configure the InfoEye system (i.e., to select push attributes, push thresholds, and push intervals). Next, we generate another set of "validation queries" according to the same model, and resolve the queries against our system configuration. The cost
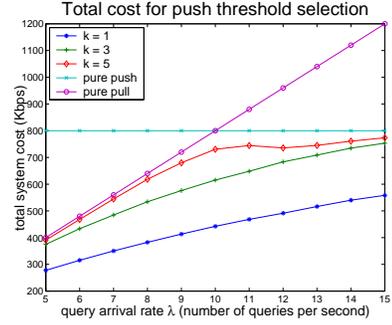

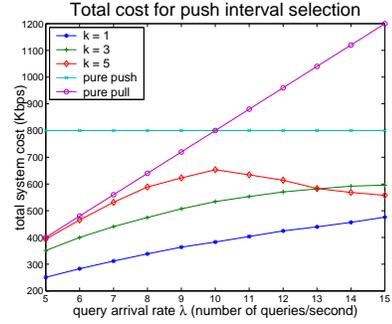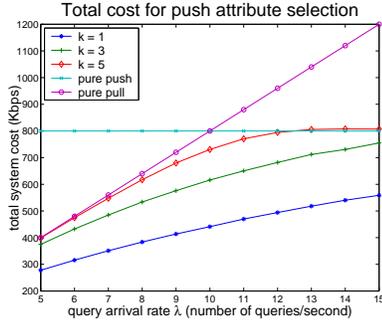Fig. 6. Push threshold selection.

Fig. 7. Push interval selection.

of the system for resolving the validation queries is computed. Each experiment is repeated 200 times, and the average cost is reported.
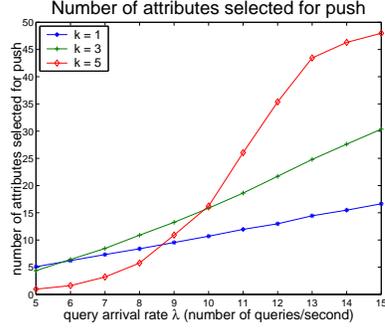
We mainly compare the system cost of InfoEye to that of the two static approaches, pure push and pure pull. In pure push-based systems, each monitoring sensor periodically reports all attribute data using the default push interval. Thus, the system cost is independent of the query arrivals. In pure pull-based systems, no periodic information push is involved. thus the system cost is proportional to the rate of query arrivals.

B. Simulation Results

Figure 5 shows the performance of our attribute selection algorithm. Figure 5(a) shows the system cost of InfoEye for different query arrival rate and maximum number of attributes k in a query. Figure 5(b) shows the number of attributes selected for push. The results show that InfoEye consistently performs better than both pure push and pull approaches. When the query arrival rate is small, pure push involves a lot of unnecessary overhead. At this time, InfoEye can configure the monitoring sensors to push only a small number of most popular attributes (as shown in Figure 5(b)), and achieve a small system cost similar to pure pull. When the query arrival rate increases, the cost of pure pull increases linearly. However, InfoEye can configure the monitoring sensors to push more attributes. As a result, its system cost is always smaller than either pure push or pure pull. If the system is statically configured, the system cost would be many times that of InfoEye for either small or large query arrival rates.

---

[6]CoMon is essentially a push based system. In order to minimize the monitoring overhead, the push interval is set to 5 minutes.

(a) Cost of multi-attribute queries.

(b) Number of attributes selected.
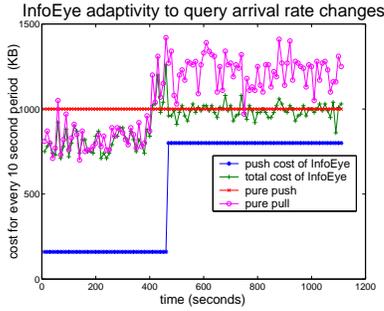
Fig. 5.   Attribute Selection Results
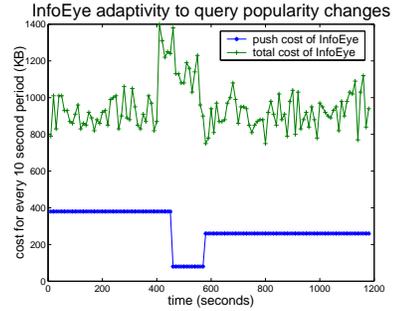

Fig. 8.   Adaptivity to query rate.

Fig. 9.   Adaptivity to popularity.

Figure 6 shows system cost when both attribute selection and push threshold selection are applied. The node attribute data are generated using the distribution described in Section IV-A, and the "moderate value" v is 5. We can see that when the query arrival rate λ is small, the cost of InfoEye is similar to Figure 5(a). This is because when λ is small, $|A^*|$ is small. As a result, the system cost is dominated by pulling attributes that are not in A^*. However, when λ is large, more attributes are pushed, and the effect of push threshold selection becomes more significant. Figure 7 shows the system cost when all three algorithms are applied. The push interval for pure push is $T = 30$ seconds. The query staleness requirement follows a distribution similar to that used for node attribute distribution. The minimum requirement is 30 seconds, the maximum requirement is 180 seconds, and the moderate value is 50 seconds. Figure 7 shows that by pushing the attributes at a frequency that satisfies most (but not all) query requirement, we can further reduce the system cost so that even when the query arrival rate is large, the total cost of InfoEye is about 25% smaller than pure push [7].

We now examine the adaptivity of InfoEye, i.e., its ability to re-configure itself in response to dynamic query pattern changes. We only show the results of push attribute reconfiguration due to the space limitation. Figure 8 shows the

[7]Figure 7 shows that when λ is large, the cost for $k = 5$ is can actually be smaller than $k = 3$. This is because for $k = 5$, more attributes are pushed as indicated in Figure 5. As a result, push interval selection has more space for improving the push cost.

adaptivity of InfoEye when the query arrival rate changes. For this experiment, initially the mean query arrival rate is four queries/second. After the initial configuration, we generate validation queries and record the total system cost every 10 seconds. An exponential weighted moving average of this "instant cost" is then compared with the system cost predicted by Equation (1). If the difference between the two costs exceeds 20%, a re-configuration is initiated. For this experiment, we also use a "historical query window" of the recent 2000 queries. System re-configuration is based on these historical queries. At time 400, we change the query arrival from 8 to 12. Figure 8 shows that the higher query arrival rate results in higher system cost. At time 470, InfoEye detects the system change and re-configures itself to push more attributes. Although push cost is increased, the total system cost is reduced since less queries need to be resolved by pull. Figure 8 also shows the cost of pure push and pull. We can see when the query arrival rate is 4, the cost of InfoEye is close to that of pure pull. Both are much smaller than pure push. After the reconfiguration, the cost of InfoEye is close to that of pure push, and both are much smaller than that of pure pull. Figure 9 shows the adaptivity of InfoEye to attribute popularity changes in the queries. The experiment settings are similar to the previous one, except the mean query arrival rate is 10 for the whole experiment. At time 400, we switch the popularity of the top three and bottom three attributes. We observe that InfoEye can quickly detect this change and reconfigure itself. Because at this time, the history queries are a mixture of

two different patterns, only a smaller number of attributes are selected. After another 120 seconds or so, most queries in the history window are from the new distribution. As a result, InfoEye reconfigures again and selects the right subset of A^* for push.

C. Prototype Results

We have implemented a prototype of our InfoEye system and deployed it on the PlanetLab [11] testbed. We have a monitoring sensor on each PlanetLab node, which can periodically check the local resource attributes and push the data to the management node. The management node is responsible for storing the pushed attribute data and answering queries. It is also responsible for running the configuration algorithms and configure the monitoring sensors based on the computed system parameters such as the push threshold for each attribute. Currently we have only integrated the push threshold selection algorithm with our management node. In addition to the monitoring sensors and the management node, we have a query client. This query client again generates synthetic queries and send the queries to the management node. The management node and query client are run on a local machine.

Our experiments involve about 280 PlanetLab nodes. Each monitoring sensor samples the local resource values every 10 seconds, and compares them with the configured push thresholds. If the resource values are greater, the attribute data are pushed to the management node. The management node accepts the pushed data and answers queries. It also invokes the push threshold selection algorithm every 60 seconds [8]. The new push thresholds are then sent to all monitoring sensors. The query client can generate queries of different patterns and send the queries to the management node. Each query specifies requirements on three attributes: available CPU time, amount of free memory, and amount of free disk space. The management node keeps a sliding window of past 1000 queries for the push threshold configuration. Each time before the configuration, the management node also runs a global aggregation query to get the node attribute distribution for the whole system. Under the above settings (e.g., 280 nodes and 1000 historical queries), each configuration run takes about 3ms, and the memory consumption of the management node is under 5MB.

For the first experiment, we first let the query client generate queries that require small amount of CPU time, free memory and disk space. Specifically, the lower bound for these attributes are randomly distributed within [10%, 20%], [10MB, 20MB] and [10GB, 20GB], respectively. After about 12 minutes, the query pattern is changed. The queries now require a minimum of CPU, free memory and disk space that are randomly distributed within [20%, 30%], [20MB, 30MB] and [20GB, 30GB], respectively. The query arrival rate is 4 per second for the entire experiment. Figure 10 shows the push

[8] System reconfiguration can be triggered by either a timer or any changes in system parameters. Our current prototype only implements the timer-triggered reconfiguration.
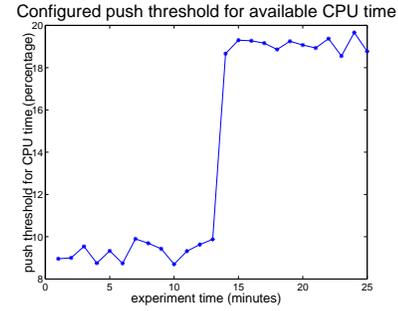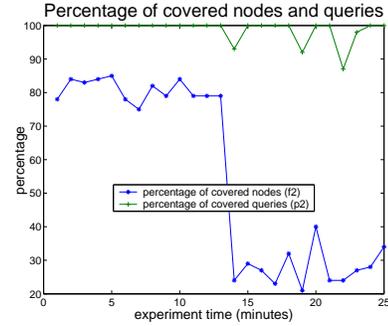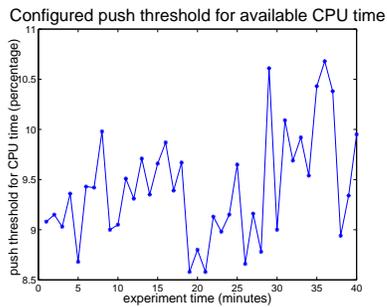

Fig. 10.   Push threshold for CPU.

Fig. 11.   Covered nodes and queries.

threshold configured by the management node every minute. Initially the push threshold for CPU time is configured to be a little less than 10%. After the pattern change, the push threshold is configured to be a little less than 20%. The push threshold for free memory and disk space show similar trend and are therefore omitted. From Figure 11 we can see the effect of such system configuration. Initially, since the push threshold is low, about 80% of the nodes need to periodically push their attributes. When the query pattern has changed and the queries require more resources, less nodes can satisfy the queries. Our push threshold selection algorithm correctly recognizes this, and configures the push thresholds to higher values. This results in only about 30% of the nodes periodically push their attribute data. Although this means a small proportion of queries $(1 - p_2)$ have to be resolved by pull, the overall system cost is reduced, due to large savings in the push cost.

Figure 12 and Figure 13 show the same results for a different query pattern change. For this experiment, during the first 15 minutes, the queries are generated just like the first experiment. Thereafter, the query distribution is not changed, but the mean query arrival rate is changed to 2. Figure 12 shows when the query arrival rate decreases, the configured push threshold for CPU is increased. The reason is that a smaller query arrival rate means a smaller overhead for query pull. As a result, the system cost can be reduced by slightly increasing the push threshold, which leads to smaller percentage of nodes that periodically push their data, and a small percentage of queries that need to invoke pull operations (as shown in Figure 13).
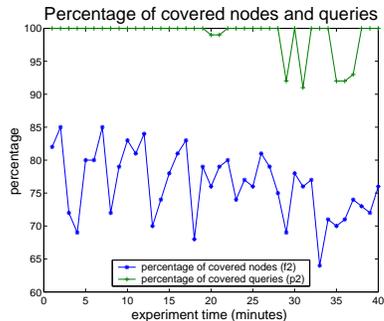
Fig. 12. Push threshold for CPU.

Fig. 13. Covered nodes and queries.

V. RELATED WORK

Distributed information management is critical for any large-scale system management infrastructure. For example, both the CoMon PlanetLab monitoring service [1] and the Grid Monitoring/Discovery Service (MDS [5]) have proven extremely useful for their user communities. However, for practical purposes, both systems are statically configured. Every node pushes all attribute data to a central server at fixed intervals, even when the attribute data are unlikely to satisfy application queries.

Astrolabe [13] and SDIMS [15] are two representative scalable distributed information management systems. Both rely on hierarchical aggregation as a fundamental abstraction. Astrolabe maintains one aggregation tree and uses gossip protocols for data reconciliation. SDIMS is built on top of a distributed hashtable (DHT) and utilizes DHT routing to build multiple aggregation trees. Data aggregation allows these systems to achieve very good scalability. However, it also means the primary focus of these systems is aggregation queries such as MIN, MAX, and SUM. In contrast, InfoEye considers multi-attribute range queries that must be resolved using detailed information about individual nodes. Several systems such as Mercury [3], SWORD [10] and PIER [8] can support multi-attribute range queries. However, their focus is on how to resolve queries in different decentralized architectures. InfoEye is complementary to these systems in that it looks at a new design dimension. The idea of utilizing statistical query patterns can potentially be applied to different architectures.

Combining push and pull based information access has been explored by some previous work in different contexts such as delivering dynamic web objects to clients [6] and collecting data in a sensor network [12]. Although the general idea of combining push and pull is not new, we should emphasize applying the idea to a specific environment requires non-trivial system analysis and design. In our case, it means identifying application query patterns and deriving the analytical model for total system cost, which makes it possible for adaptive push/pull configuration.

Wills et. al. [14] have considered the problem of adaptive resource information management. However, they assume an environment where multicast is used to disseminate information requests or updates, and the goal is to decide whether requests or updates should be multicast, based on the local observation of each node in the system. Thus it is also different from our system.

VI. CONCLUSION

We have presented the design and evaluation of InfoEye, a novel model based, self-adaptive distributed information management system. The goal of InfoEye is to resolve multi-attribute queries in large-scale dynamic distributed systems with minimum monitoring overhead. To achieve this goal, InfoEye maintains statistical information about both application queries and node attribute distributions, and dynamically configure itself to achieve minimum management overhead. Through extensive simulation studies, we show that InfoEye can achieve much lower management overhead than static solutions. More importantly, when the query pattern changes, InfoEye can quickly re-configure itself to adapt to the changes. We have also implemented a prototype of the InfoEye system and validated the feasibility and performance on a real network environment. A web based query interface is also provided at http://cairo.cs.uiuc.edu/monitoring/.

Our current InfoEye system can be seen as a first step to demonstrate the feasibility of adaptive information management. As ongoing work, we are considering applying the same idea to overlay link monitoring, which is important for many distributed networked applications.

REFERENCES

[1] CoMon. http://comon.cs.princeton.edu/.
[2] World Community Grid. *http://www.worldcommunitygrid.org*.
[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *SIGCOMM 2004*, August 2004.
[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *IEEE Infocom 1999*, 1999.
[5] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *The 10th IEEE Symposium on High Performance Distributed Computing (HPDC10)*, 2001.
[6] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. In *WWW10*, May 2001.
[7] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *International J. Supercomputer Applications*, 2001.
[8] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proceedings of 29th VLDB Conference*, 2003.

[9] J. Liang, S. Y. Ko, I. Gupta, and K. Nahrstedt. Mon: On-demand overlays for distributed system management. In *Second Workshop On Real, Large Distributed Systems (WORLDS'05)*, December 2005.

[10] D. Oppenheimer, J. Albrecht, D. Patterson, and A. Vahdat. Design and implementation tradeoffs for wide-area resource discovery. In *HPDC-14*, July 2005.

[11] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *1st Workshop on Hot Topics in Networks (HotNets-I)*, Princeton, New Jersey, October 2002.

[12] N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Hybrid push-pull query processing for sensor networks. In *Workshop on Sensor Networks at Informatik*, 2004.

[13] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalabel technology for distributed system monitoring, management, and data mining. *ACM Transactions on Computer Systems*, 21(2):164–206, May 2003.

[14] C. E. Wills and S. Chandra. Adaptive resource management, 1995.

[15] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. *Proc. of SIGCOMM 2004*, Aug. 2004.