Optimization Techniques for Reactive Network Monitoring

Ahmet Bulut, *Member, IEEE,* Nick Koudas, *Member, IEEE,* Anand Meka, Ambuj K. Singh *Member, IEEE,* and Divesh Srivastava *Member, IEEE*

Abstract—We develop a framework for minimizing the communication overhead of monitoring global system parameters in IP networks and sensor networks. A global system parameter is defined as a function of local properties of different network elements. Identifying when the total amount of interface out traffic from an organization's sub-network exceeds some threshold is an example parameter to monitor. Our main idea is to optimize the scheduling of local event reporting across network elements for a given network traffic load and local event frequencies. Our system architecture consists of N distributed network elements coordinated by a central monitoring station. Each network element monitors a set of local properties, and the central station is responsible for identifying the status of global parameters registered in the system. We design an optimal algorithm when the local events are independent; whereas, when they are dependent, we show that the problem is NP-complete and develop two efficient heuristics: the *SPA* (*S*ample, *P*artition, and *A*ggregate), and *Ada* (*Ada*ptive) algorithms which adapt well to changing network conditions, and outperform the current state of the art techniques in terms of communication cost.

Index Terms—Network Monitoring, Push Pull techniques, Bayesian Networks

1 Introduction

Reactive network monitoring consists of measuring the properties of the network to ensure that the system operates with desirable parameters. The management station queries the state of the network in order to react to alarm conditions that may develop in the network [7]. Information about the network state is collected using two different techniques: event reporting and polling. In event reporting, network elements distributed across the network push alarms and detailed event reports to the station. In polling, the station sends requests to obtain the status of network elements. Typically, polling is done periodically with a fixed frequency, determined by a critical time window within which the alarm condition has to be detected.

In many situations, there is a need to monitor a global system parameter, which is defined as a function of local properties of different network elements. In such cases, after detecting local changes, each network element has to continuously emit alarms in order to ensure that global parameters are not violated. In sensor networks, a typical example is a monitoring system which determines whether the average temperature of a particular region exceeds a certain threshold. In IP networks, consider the monitoring of the amount of traffic from an organization subnetwork to the Internet. A subnetwork is connected to the outer world via a number of interfaces. The goal is to determine whether the total outbound traffic exceeds a predefined threshold. One solution to this problem, referred to as *all-pull* scheme, is to poll the status of network elements continuously. As long as the cumulative sum is below the threshold, no alarms are generated. Another approach to the same problem is to allocate a fixed budget (a small proportion of the threshold) to each node. Each time the amount of local traffic exceeds the local budget, a report containing event details and some application specific information is sent to the station. We refer to this solution instance as *all-push*.

Given a set of n network elements, where the decision at each element is either "to push" or "to pull", one can observe that the solution space of schedules is exponential in the number of elements (2^n), and the *all-push* and *all-pull* schemes correspond to two specific solutions in this space. The major disadvantage of the all-pull and all-push schemes is that they are oblivious to the environment characteristics. All-pull scheme does not take local event frequencies into account, and incurs cost continuously especially if the transmission network is congested. All-push scheme considers this aspect, but since it functions on local information only, there is no global coordination. An efficient approach to tackle this problem is to combine event reporting with aperiodic polling such that only a subset of elements are chosen as watch-dogs for monitoring a given global parameter. When reports from all watch-dogs are received, then the status of remaining elements are obtained using polling. Therefore, the problem of interest becomes how to select the set of elements that will push. A simple greedy heuristic, which selects to push from elements with a low event frequency, performs very well in practice. One has to identify the top-k least frequent event set continuously.

Our main idea is to formulate the cost of monitoring multiple parameters using the information about the statistical

A. Bulut is with Citrix Online Inc, 5385 Hollister Ave, Santa Barbara, CA 93111. E-mail: bulut@citrix.com.
N. Koudas is with the Department of Computer Science, Bahen Center for Information, University of Toronto, 40St. George Street, Rm Ba5240, Toronto ON M5S 2E4. Email: koudas@cs.toronto.edu
A. Meka and A. K. Singh are with the Department of Computer Science, University of California Santa Barbara, Santa Barbara, CA 93106-5110. E-mail: {meka, ambuj}@cs.ucsb.edu.
D. Srivastava is with AT&T Labs-Research, 180 Park Ave., Bldg. 103, Florham Park, NJ 07932-0971. E-mail: divesh@research.att.com.

characterization of the whole set of network elements, e.g, the frequency of occurrence $p(e)$ associated with each local event e, and the cost of a message containing event details or pull requests. For each event, the binary scheduling decision is either push or pull. We have three types of messages in the system: (1) push message, (2) pull request, and (3) answer to a pull request. The cost of each message may depend on many parameters such as the size of the message and the load on the specific network the message traverses. In order to model communication cost in the most general scenario, we use different costs (C_j) for each message type in our framework. This offers the ability to experimentally study the relative tradeoffs. Our algorithms compute the cost of each schedule as a function of $p(e)$'s and C's. The schedule with the least cost is selected for the current environment. However in reality, system characteristics are dynamic and change over time. Therefore, the least costly schedule is also expected to change. In that sense, our techniques perform a continuous optimization in the solution space, and adapt to the current environment.

Our contributions in this paper are:

1) We formulate the problem of minimizing the communication cost of monitoring a global aggregate over a set of local events as an optimization problem.
2) We design an optimal solution when the events are independent and show that the optimization problem is NP-complete when dependencies are introduced between events.
3) When correlations exist within the event set, we propose two efficient algorithms, the *SPA* and the *Ada* algorithms. The SPA algorithm performs an off-line summarization of schedules using partial costs, and thus determines low-cost schedules on the fly with negligible computational cost; whereas Ada performs a greedy search for the optimal in an efficient manner and re-optimizes in the dynamic case only when thresholds are violated.
4) We perform experiments on both real-world and synthetic data sets and show that both Ada and SPA algorithms outperform the competing techniques by two orders of magnitude in communication costs with a tolerable computational overhead.

1.1 Outline of the Paper

The remainder of the paper is structured as follows: we first introduce preliminaries for our monitoring framework in Section 2. We derive a cost model for scheduling a set of events, and analyze statistical characteristics of the system and their implications on the cost formulation. In Section 3, we discuss the hardness of the optimization problem in different network settings. In Section 4, we consider various optimization techniques to solve our problem, and present algorithms to realize them in our setting . In Section 5, we present the results of an extensive set of experiments in order to study the effectiveness of our approaches. Finally, we discuss avenues for future work and conclude in Section 7.

2 Formal Model

2.1 Event model

Consider a network with n elements. Let N_i denote the i^{th} network element. In our monitoring framework, each network element N_i monitors a function f over its stream of data. Time t is an integer, beginning at $t = 1$. Let the stream of data inspected by N_i until time t be $x_i^1, \ldots, x_i^{t-1}, x_i^t$. The window of the k most recently seen data values on a stream is denoted by $w_i : (x_i^{t-k+1}, \ldots, x_i^t)$. Function f is defined over such a window w_i. At time t, if the value of the function $f(w_i)$ exceeds the threshold τ_i, a corresponding local event occurs. The function f we use is an application specific input parameter. Aggregates such as sum, $spread$, i.e., $max - min$, and $count$ are some example functions.

The threshold values τ_i of each network element N_i can either be specified as part of the input, or they can be determined using historical data. For example, in a networking application scenario a router can specify tolerance levels for packet drops; such levels constitute a threshold value. By definition, an event at some window w_i occurs if the aggregate value computed on that window deviates significantly from most of the aggregate values computed on windows of the same size. One way of setting the threshold τ_i is $g_i(\mu, \sigma)$, where g_i is a linear function of the mean μ and the standard deviation σ of historical $f(w_i)$ values.

Definition 2.1: A local event at network element N_i is a random variable r_i such that

$$r_i = \left\{ \begin{array}{ll} 1 & \text{if } f(w_i) \geq \tau_i \\ 0 & \text{otherwise} \end{array} \right\} \quad (1)$$

Definition 2.2: The probability of occurrence $p(r_i)$ of a local event r_i corresponds to the likelihood of the event occurring in the window w_i.

At the monitoring station, global parameters are defined in terms of a subset of the local events. An alarm on the global parameter is generated if *all* of the respective events occur.

Definition 2.3: A set of local events together defines a global parameter, and is referred as a "query" in the rest of the paper.

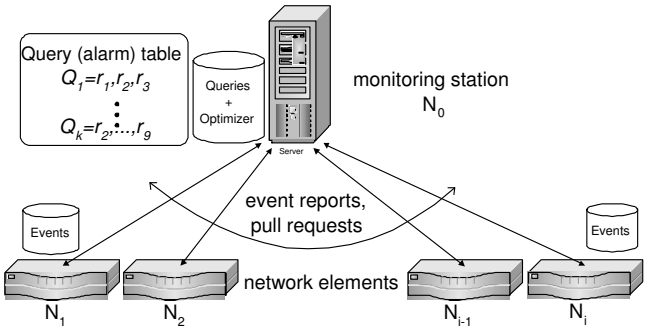


Fig. 1. Event-driven monitoring framework for a network consisting of one monitoring station and a set of network elements.

Figure 1 shows our framework. Each network element N_i monitors a local event r_i. At the monitoring station N_0, users register queries that are defined in terms of a set of local

events. The communication between the monitoring station and the network elements are messages that contain event reports or pull requests, i.e., status checks. This paper is targeted towards optimizing the communication cost in the case of a single query. Optimization over multiple queries is addressed in [**?**].

2.2 Communication cost model

The events are propagated between the monitoring station and the network elements in two modes: The network element can either **push** the event to the station incurring a cost of C_1. The monitoring station can also poll (**pull**) a network element for the existence of an event r_i incurring a cost of C_2. If the event has occurred, the network element replies back incurring a cost of C_3. As noted earlier, the cost of each message can vary across applications. Therefore, we use different costs for generality and to study tradeoffs. We assume *reliable communication* and *global time synchronization*. In the rest of the paper, we simply use "event" to refer to a local event. We illustrate all of our concepts with an example configuration below.

Example 1: Assume three network elements N_1, N_2, and N_3 each monitoring a single event r_1, r_2, and r_3 respectively. Let the associated probabilities be $p(r_1)$, $p(r_2)$, and $p(r_3)$ respectively. There is a single query Q_1, a conjunctive predicate on events r_1, r_2 and r_3 registered at the monitoring station. Since for each event we can have two possible ways of communication (push or pull), there are 2^n schedules for n events. Assuming 1 is a push and 0 a pull, we have eight possible schedules for $n = 3$ events as $000, 001, 010, 011, 100, 101, 110$, and 111. Among these schedules, $S = 000$ corresponds to all-pull, while $S = 111$ corresponds to all-push.

Let $\mathcal{R}$ denote the set of all local events. We use $\mathcal{R}^+$ to denote the set of events in Q_1 that are pushed, and $\mathcal{R}^-$ to denote the set of events that are pulled in a given schedule S. Each event r in $\mathcal{R}^-$ is pulled only if all events in $\mathcal{R}^+$ occur, which happens with probability $p(\cap_{r_i \in \mathcal{R}^+} r_i) = p(\mathcal{R}^+)$. The answer to this pull request occurs with probability $p(r|\mathcal{R}^+)$, which is the conditional probability of occurrence for r given that all push events occurred. For example, let the schedule S be 010. Then, the sets for query Q_1 are $\mathcal{R}^+ = \{r_2\}$ and $\mathcal{R}^- = \{r_1, r_3\}$. We can formulate the cost $C(S)$ of S in terms of the probabilities $p(r_i)$'s, and the cost parameters C_1, C_2, and C_3 as follows:

$$C(S) = \overbrace{p(r_2)C_1}^{pushcost} + \overbrace{p(r_2)(C_2 + p(r_1|r_2)C_3)}^{pullcost\ for\ r_1} + \underbrace{p(r_2)(C_2 + p(r_3|r_2)C_3)}_{pullcost\ for\ r_3}$$

where each individual term denotes the expected cost:

$p(r_2)C_1$	an event report (push) on r_2
$p(r_2)C_2$	a pull request initiated as a result of r_2
$p(r_1\|r_2)C_3$	an answer r_1 conditioned on r_2
$p(r_3\|r_2)C_3$	an answer r_3 conditioned on r_2

We are interested in minimizing the total communication cost required to detect alarm conditions (queries) specified by users. Therefore, we mainly consider communication complexity. We state the main optimization problem we consider within the scope of this paper as follows:

Problem 1: Given the event probabilities $p(r_i)$'s that are functions of time, and the communication cost parameters C_1, C_2, and C_3, identify at all times the optimal schedule in terms of communication cost.

2.3 Cost model for event scheduling

The cost of a schedule is the sum of the total push cost and the total pull cost. We first consider the case of assuming statistical independence.

2.3.1 Independence Case

If all the events are mutually independent, an answer to a pull request for an event r in $\mathcal{R}^-$ occurs with probability $p(r|\mathcal{R}^+) = p(r)$. Then, the cost $C(S)$ for schedule S can be expressed as:

$$\sum_{r_i \in \mathcal{R}^+} p(r_i)C_1 + \prod_{r_i \in \mathcal{R}^+} p(r_i) * \left(\sum_{r \in \mathcal{R}^-} C_2 + p(r)C_3 \right)$$

where the first term is the total push cost, and the second term is the total pull cost. Ideally, the cost model should take into account all dependencies between the events being monitored. The dependencies can either be *intra*-dependencies that arise at a given network element due to the nature of the aggregate function used [10], [15], or they can be *inter*-dependencies that arise due to the structure of the network being monitored.

2.3.2 Conditional dependence

When we take statistical dependencies into account, the threshold computation becomes more complicated than that of assuming statistical independence. For a given schedule S, the cost $C(S)$ is equal to:

$$\sum_{r_i \in \mathcal{R}^+} p(r_i)C_1 + p(\mathcal{R}^+) * \left(\sum_{r \in \mathcal{R}^-} C_2 + p(r|\mathcal{R}^+)C_3 \right)$$

Example 2: The cost $C(S)$ of $S = 1100$ for the query configuration $Q_2 : \{r_1, r_2, r_3, r_4\}$ is:

$$= \overbrace{p(r_1)C_1 + p(r_2)C_1}^{pushcost\ of\ \mathcal{R}^+} + \overbrace{p(r_1, r_2)(C_2 + p(r_3|r_1, r_2)C_3)}^{pulling\ for\ r_3^-} + \underbrace{p(r_1, r_2)(C_2 + p(r_4|r_1, r_2)C_3)}_{pulling\ for\ r_4^-}$$

where we have some cost terms involving multivariate probabilities.

The storage space required for representing a multivariate probability distribution $p(\mathbf{q})$ of n discrete random variables is enormous. In our case, each event r_i is a binary random variable; therefore, the random vector $\mathbf{q} = (r_1, r_2, \ldots, r_n)$ can take as many as 2^n values. Assuming that $p(\mathbf{q})$ is unknown and that s independent samples $q^1, q^2, \ldots, q^s$ are available, the complete specification of $p(\mathbf{q})$ can be expressed in $\Theta(2^n)$

space [20]. However, the amount of space allowed for storage is limited, and the number of available independent samples is usually small. Therefore, the best one can do is to approximate $p(\mathbf{q})$ with some simplifying assumptions. A method for optimal approximation of an n-variate discrete probability distribution using first order dependence relationship, was considered by Chow and Liu [3]. For the rest of the paper, we assume that the probability distribution is defined by a first-order dependence tree; such a tree is a specific case of a Bayesian network [11]. The approximation method by Chow and Liu is discussed in Appendix A. Given such a dependence tree, the following example shows the probability computation for a set of events in different cases.

Example 3: Consider Figure 2. The probability distribution over the events $r_1, \ldots, r_6$ is approximated by two disjoint first-order dependence trees. Therefore, an event belonging to the set $\{r_1, r_2, r_3\}$ is independent of any event in the set $\{r_4, r_5, r_6\}$. Now consider the first tree. The event r_2 is conditionally independent of r_3, given r_1, i.e., $p(r_2|r_1, r_3) = p(r_2|r_1)$.

From the figure, we notice that the prior probability at the root r_1 is $p(r_1) = 0.4$, and the conditional probability of event r_2 given the event r_1 is $p(r_2|r_1) = 0.3$; whereas, the conditional probability of event r_2 given the event $\bar{r}_1$ is $p(r_2|\bar{r}_1) = 0.2$. We can compute the probability of event r_2, $p(r_2)$ by employing the principle of Mutual Exclusion (ME) followed by the Bayes rule [**?**]:

$$\begin{aligned} p(r_2) &= p(r_1, r_2) + p(\bar{r}_1, r_2) \\ &= p(r_1)p(r_2|r_1) + p(\bar{r}_1)p(r_2|\bar{r}_1) \\ &= 0.4 \times 0.3 + 0.6 \times 0.2 = 0.24 \end{aligned}$$

The joint probability of a set of events (r_2, r_3), $p(r_2, r_3)$ is calculated using the the notion of conditional independence along with the ME and Bayes principles:

$$\begin{aligned} p(r_2, r_3) &= p(r_1, r_2, r_3) + p(\bar{r}_1, r_2, r_3) \\ &= p(r_1)p(r_2|r_1)p(r_3|r_1, r_2) \\ &\quad + p(\bar{r}_1)p(r_2|\bar{r}_1)p(r_3|\bar{r}_1, r_2) \\ &= p(r_1)p(r_2|r_1)p(r_3|r_1) + p(\bar{r}_1)p(r_2|\bar{r}_1)p(r_3|\bar{r}_1) \\ &= 0.4 \times 0.3 \times 0.1 + 0.6 \times 0.2 \times 0.4 = 0.06 \end{aligned}$$

Since r_4 is independent of r_2 and r_3, the joint probability of the set of events (r_2, r_3, r_4), $p(r_2, r_3, r_4)$ is:

$$\begin{aligned} &= p(r_2, r_3)p(r_4) \\ &= 0.06 \times 0.3 = 0.018 \end{aligned}$$

When the dependency graph has cycles, the above probability computations are NP-hard [20]. However, for first-order dependence trees, any marginal probability, i.e., $p(r_i)$ can be computed in $O(n)$ time, using Pearl's message passing algorithm [20].

3 Scheduling Problem: Optimality and Complexity

In this section, we present techniques to solve the optimization problem in two different settings. When the events at each node are independent, we design a polynomial time optimal solution; whereas when the events are dependent, the problem is shown to be NP-complete.

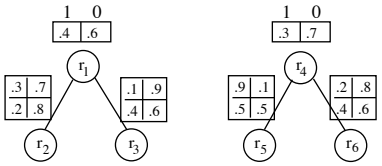


Fig. 2. An example of first-order dependence trees.

3.1 Independence case

We assume *a priori* knowledge on $p(r_i)$'s and that the events are mutually independent of each other. Assuming that such probabilities are fixed, our algorithm first partitions the 2^n schedules, into $n + 1$ disjoint classes. For each class, we identify the schedule that achieves the minimum cost, and then find the optimal over all classes.

The whole space of schedules S is partitioned into $n + 1$ classes $\chi_0 \ldots \chi_n$. A schedule $s \in \chi_k$ has exactly k pushes. Consider Example 1 in Section 2.2. Assume for simplicity that $C_1 = C_2 = C_3 = 1$. Let $p(r_1), p(r_2)$, and $p(r_3)$ be equal to $0.5, 0.3$, and 0.4 respectively. The costs associated with each possible schedule and their corresponding classes are shown in Table 1.

S	Cost Expression	C(S)
χ_0		
000	(1+0.5)+(1+0.3)+(1+0.4)	4.20
χ_1		
001	0.4+0.4(1+0.5)+0.4(1+0.3)	1.52
100	0.5+0.5*(1+0.3)+0.5*(1+0.4)	1.85
010	0.3+0.3*(1+0.5)+0.3*(1+0.4)	1.19
χ_2		
101	0.5+0.4+0.5*0.4(1+0.3)	1.16
110	0.5+0.3+0.5*0.3(1+0.4)	1.01
011	0.3+0.4+0.3*0.4(1+0.5)	0.88
χ_3		
111	0.5+0.3+0.4	1.20

TABLE 1
Partitioning of schedules and corresponding costs for the scenario in Example 1.

Notice that the winner in class χ_1 is schedule 010, where the event r_2 with the smallest probability (0.3) is set to push. Similarly, the winner in class χ_2 is 011 which corresponds to the two events with the smallest probability being set to push. The minimum over all classes is calculated as 011.

Our algorithm functions as follows: Given the set of input probabilities over events r_j, for $j = 1$ to n, we first sort the probabilities in a non-descending order, and rename the events $r_1 \ldots r_n$ such that $p(r_1) \leq p(r_2) \leq p(r_3) \ldots p(r_{n-1}) \leq p(r_n)$. In each class χ_k, the set of k events with the minimum probability, i.e., the events corresponding to $r_1 \ldots r_k$ are set to push. Theorem 3.1 shows that this schedule is indeed the minimum over class χ_k. Then the optimal is calculated as the minimum over the $n + 1$ classes. In the case of independent events, the cost of computing a schedule is $O(n)$. Therefore, the computational complexity of the algorithm is $O(n^2)$.

Theorem 3.1: Given the costs C_1, C_2 and C_3 and (sorted) events $r_1 \dots r_n$ such that $p(r_1) \leq p(r_2) \leq p(r_3) \dots \leq p(r_n)$, the schedule $s* \in \chi_k$ in which the events corresponding to $r_1 \dots r_k$ are set to push is the optimal schedule in class χ_k.

Proof: Consider schedule $s*$ and an arbitrary schedule $s_1 \in \chi_k$ with the k push bits permuted. For simplifying the discussion, assume that there is a disparity of two bits between $s*$ and s_1. The proof can easily be generalized for any $s_1 \in \chi_k$. The cost of the schedule $s*$

$$= \overbrace{C_1 \sum_{i=1}^{k} p(r_i)}^{cost\ of\ \mathcal{R}^+} + \overbrace{p(r_1)...p(r_k) \sum_{j=k+1}^{n} (C_2 + p(r_j)C_3)}^{pulling\ for\ R^-}$$

The above expression can be decomposed into three parts: costs involving the C_1 term, the C_2 term and the C_3 term. Similarly, we can split the costs of s_1. First, we compare the C_1 costs of $s*$ and s_1. Since, $s*$ selects the k smallest $p(r_i)$'s, the total C_1 costs of $s*$ is at most the cost of s_1. Similarly, we can show that the total C_2 cost of $s*$ is at most the cost of s_1. While comparing C_3 costs, we can show that each individual term of $s*$ is smaller than the corresponding term of s_1. □

3.2 Conditional Dependence

In this section, we formally show that this problem becomes computationally intractable when joint dependencies are introduced between pairs of nodes.

Theorem 3.2: Given a joint distribution represented by a first-order dependence (Bayesian) tree, the decision problem: *"Does there exist a schedule $s \in \chi_k$ with cost at most c ?"* is NP-complete.

Proof: We outline the proof here. Our problem can be reduced from the 0-1 Integer Programming problem [9], with a variable set to 1 if the corresponding event is set to push, and 0 otherwise. The minimization function is a polynomial of degree n, with the constraint that the sum of the n variables adds to k. Interested readers can refer to the technical report [**?**] for details. Unfortunately, the problem remains intractable even when the Bayesian tree degenerates to a forest of edges. □

4 Scheduling Algorithms

The hardness results in the previous section indicate that a polynomial time algorithm is unlikely. In this section, we present efficient algorithms which not only generate schedules with small cost but are also designed to minimize the computational complexity in the dynamic case.

A simple solution to the optimization problem is to draw a large set of schedule samples, compute the cost of each schedule, and ascertain the schedule with the minimum cost. But, when the event probabilities change, such a solution would necessitate an expensive recomputation of the costs of each schedule in the huge sample space. The first technique, the *SPA* algorithm is primarily designed to overcome the above computational burden of recomputing a solution, by computing several partial costs for a schedule which can be reused in the dynamic case.

The second technique, *Ada* starts with a schedule, and progressively generates schedules with a smaller cost in a greedy manner; finally, resulting in a schedule with minimal cost. When the underlying data distribution changes, Ada identifies potentially beneficial time instances for re-optimization, and follows an adaptive optimization procedure.

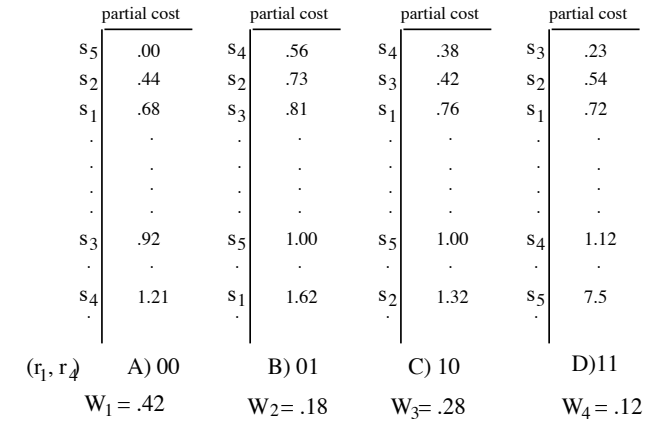


Fig. 3. SPA algorithm: Ranked list for each combination of root values. The root priors shown in Figure 2 are employed to calculate the weights W_i of each list.

4.1 *SPA*: Algorithm

The *SPA* algorithm is targeted for optimization over a specific class of applications, in which the first-order dependence network is decomposed into a disjoint set of trees. A justifying case is the recent work in sensor networks [4], which decomposes the sensor network into a disjoint set of spatial clusters and maintains a probabilistic model for each cluster of correlated nodes. In IP networks, a group of routers observing similar traffic patterns might be characterized by a single tree; and hence the whole network can be represented by a set of disjoint trees. In the dynamic case, we exploit the strong intra-cluster correlations in such settings, and assume that only the prior probabilities at the roots of each tree change, while the conditional probability tables at each internal edge remain intact. This fact is exploited to pre-process a large amount of data, and only recompute when necessary.

Unlike the simple solution, the SPA algorithm precomputes several reusable partial costs for each schedule; a partial cost corresponding to each combination of root values, and then calculates the total cost of a schedule as the weighted sum of partial costs, where the weight corresponds to the probability of occurrence of underlying combination of root values. Hence, when the priors at the root change, only the weights are altered leaving the partial costs unchanged.

Consider the example of such a total cost computation as shown in Figure 3. Assume that the probabilistic network is as shown in Figure 2. Each list shown in Figure 3 corresponds to a particular combination of root values. Consider the schedule s_5 : 100100. While computing its partial cost in list A), we assume that roots r_1 and r_4, each take the binary value 0 with a probability 1. The weight of list A) is $W_1 = p(r_1 = 0)p(r_4 = 0) = (0.6)(0.7) = 0.42$.

Algorithm 1 SPA(m *sorted sources*, $Weight\ of\ sources$)

begin procedure
Do sorted access across m sources simultaneously;
$current = \infty$;
If a schedule s is seen under a source, do a random seek in other sources and compute the total cost $T(s)$;
$current := \min(current, T(s))$;
$threshold := 0$;
for each source i **do**
$x_i :=$ the last partial cost seen under sorted access;
$threshold += W_i * x_i$; // W_i is weight of source i
end for
if $current \leq threshold$ **then**
Halt and output the current best schedule;
end if
Continue sorted access and repeat the procedure;
end procedure

Similarly, each schedule's partial cost is calculated in every list. Further, the lists are sorted to aid the computation of the minimum cost schedule. The total cost of schedule s_5 can be computed as $(0.42)(0.0) + (0.18)(1.0) + (0.28)(1.0) + (0.12)(7.5) = 1.36$. This cost will be always equal to the value obtained from the straightforward computation in Section 2.3.2 $(0.4 + 0.3 + 0.12 * 5.5 = 1.36)$.

Henceforth, we refer to each list corresponding to a particular combination of root values as a *source*. Now, the schedule with the minimum cost is obtained by a modified adaptation of the Fagin et al. [8]'s algorithm as shown in Algorithm 1. The Threshold algorithm aggregates the cost over all sources, and is proved to be instance optimal in inspecting a small number of candidates while determining the minimum cost schedule. While accessing each list simultaneously, the algorithm computes the total cost of a schedule appearing at the top of each list, and retains the current minimum. Further, it computes a *threshold*, or a lower bound on the minimum cost schedule, and halts list traversal if the current minimum is smaller than threshold.

However, memory becomes a bottleneck when we attempt to store the entire set of samples —(schedule, partial cost) tuples— for each source. Hence, further summarization of the samples at each source is required. Given a memory constraint of D tuples per source, we resort to partitioning the set of samples into *Mutually Exclusive and Collectively Exhaustive* (MECE) classes. Each class is summarized by a mean cost and the classes are selected such that they minimize the global mean squared error resulting from all the classes. We next explain the class selection.

4.2 Class Partitioning

Partitioning the set of samples into different classes offers a major benefit of reducing the memory allocation, and consequently resulting in small online computation time. However, this does come at the slight expense of decrease in quality of the resulting answer set. Therefore, this technique can be seen as a memory-quality or time-quality tradeoff.

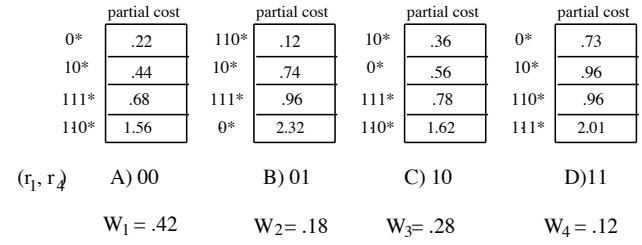


Fig. 4. Each source is partitioned into MECE classes.

Every source partitions the set of schedules into MECE classes. Consider source A in the example depicted in Figure 4. Each class represents a regular expression. Here, the partial cost refers to the mean cost of samples belonging to that class. For example, schedule 100100 belongs to the class $10*$. Cost aggregation across sources is now carried over classes rather than on schedules. Threshold algorithm is invoked to ascertain the minimum cost class, and outputs a random schedule from this class. Throughout this explanation, we assume that all the sources have the same regular expression classes[1] and present a technique to determine such a partitioning.

Given a set of samples across s sources, and a total memory budget B, we employ a decision tree based classification of samples into $D = B/s$ classes. The quality of the tree generated is measured by the global mean squared errors (GMSE) present over the D leaf classes. Realizing that building the optimal decision tree is NP-complete [11], we employ a greedy algorithm. The algorithm follows an iterative procedure, and in each iteration greedily grows the tree to maximize the reduction in the GMSE of the current tree. Next, we explain the algorithm.

Initially the decision tree has a single node (class), the regular expression $*$ representing the whole space of schedules. Assume that the algorithm is run for K steps resulting in a tree with K leaf nodes. We describe the next step involved in this iterative procedure. Each leaf node is evaluated to find the locally optimal binary split point, i.e., the split which minimizes the sum of least squared errors in the resulting child nodes. Note that each leaf node evaluated in the split is a regular expression. Each variable r_j in the LHS of the expression below is either unassigned $''.''$, or is assigned a definite value in $\{0, 1\}$ because of splits at the higher levels of the tree.

$$GreedySplit(r_1, .., r_i, .., r_n) = \min_{unassigned\ \ i} (MSE(r_1, .., 0, .., r_n) + MSE(r_1, .., 1, .., r_n))$$

Finally, among all the K nodes, that node whose split maximizes the difference in the current MSE error at the node and the sum of MSE errors of children is selected for partitioning. This procedure is initialized with $K = 1$ and is iterated until $K = D$. The L_∞ error metric can also be employed while

1. However, if that were not the case, we conjecture that the aggregation of schedules over multiple sources, each with a different class partitioning is NP-hard. The complexity arises due to the combinatorial explosion involved in the intersection of a set of regular expressions generated from each source to determine a common schedule. Therefore, we adhere to uniform partitioning across all sources.

building the decision tree, but it discards the error from every sample except for the maximum deviation. On the other hand, the L_2 error takes into account the error contribution from each sample.

4.3 *Ada*: Algorithm

In this section, we employ a different greedy algorithm *Ada* which trades computational power for huge savings in communication costs. Ada performs a greedy search for the optimal solution sin the schedule space, employing a hill-climbing [11] based technique. It also incorporates an efficient thresholding scheme to avoid the expensive process of re-optimizing the solution in the dynamic case. Next, we describe Ada's search algorithm.

Initially, a schedule is randomly chosen and its cost is measured. If by toggling an event from push to pull (or vice-versa) in the current schedule, a better cost is realized then the resulting schedule is chosen as a candidate. Over all the events, a set of candidates is generated for the next step. The candidate with the smallest cost in the set is chosen as the current schedule. This process is iterated until the solution cannot be improved any further; or, the algorithm has reached a locally optimal state. The algorithm is repeated with different seed schedules, and the solution with the smallest cost is output. The greater the number of runs the higher the probability of convergence to the globally optimal solution. This algorithm can be seen as a trade-off between the computational cost and the quality of the solution.

When the underlying data distribution changes, the current operating schedule may become sub-optimal and might necessitate an expensive re-optimization procedure. In this section, we discuss how to avoid such expensive online computational costs by introducing thresholds on each parameter.

At a given state of operation, we maintain estimates for each $p(r_i)$. Bits (push/pull events) are toggled by changes in $p(r_i)$'s associated with events. For this purpose, we identify thresholds β_{r_i} for each $p(r_i)$ at the current schedule state, such that if a $p(r_i)$ goes above or below its threshold, we change the mode of operation from push to pull or vice versa. We obtain these thresholds by analytically deriving equations on $p(r_i)$'s between the current schedule and a set of neighboring schedules which differ in a single bit.

4.4 Threshold Setting

We first explain threshold setting when the events are mutually independent. Consider Example 1 shown in Table 1. The optimal schedule $s_1 = 011$. Then the schedule change to $s_2 = 111$ is conditioned on r_1, and is triggered if $C(s_1) = C(s_2)$ when $p(r_1) = \beta_{r_1}$.

$$\begin{aligned} C(s_1) &= p(r_2) + p(r_3) + p(r_2)p(r_3)[1 + p(r_1)] \\ &= 0.3 + 0.4 + 0.3 \times 0.4(1 + \beta_{r_1}) \\ C(s_2) &= p(r_1) + p(r_2) + p(r_3) \\ &= \beta_{r_1} + 0.3 + 0.4 \\ \therefore \beta_{r_1} &= \frac{0.3 \times 0.4}{1 - 0.3 \times 0.4} \approx 0.14 \end{aligned}$$

We can generalize the above threshold setting scheme, for the case when the events are conditionally dependent. Again, we allow only the priors at the root of each dependence tree to change. Even though the conditional probability tables remain intact, the marginal probability of any other child (event) is altered by updates to priors. The following example describes the thresholding scheme.

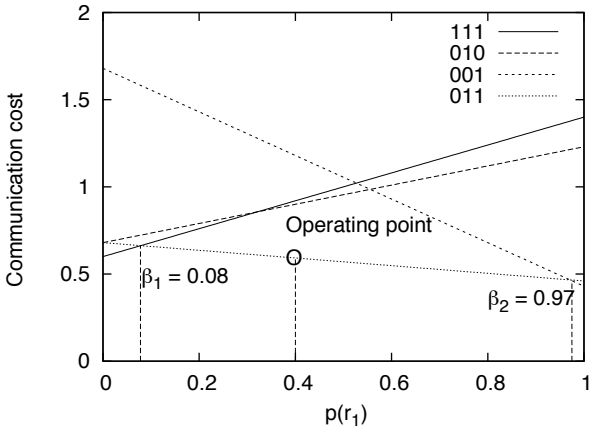


Fig. 5. Communication cost as a function of root probability $p(r_1)$. Thresholds β_1 and β_2 are set on $p(r_1)$ in order to toggle the mode of the operation.

Consider a query with three events r_1, r_2 and r_3. Let their joint distribution be as shown in Figure 2. Assume that the current optimal schedule is $s_1 = 011$. All the thresholds are set on root r_1's prior $p(r_1)$ since it is the only free variable. Assuming that the rest of probabilities do not change, a transition from schedule $s_1 = 011$ to schedule $s_2 = 111$ is conditioned on r_1, and is triggered if $C(s_1) = C(s_2)$ when $p(r_1) = \beta_{r_1}$.

$$\begin{aligned} C(s_1) &= p(r_2) + p(r_3) + p(r_2, r_3)[1 + p(r_1|r_2, r_3)] \\ &= p(r_2) + p(r_3) + p(r_2, r_3) + p(r_1, r_2, r_3) \\ &= p(r_1)[p(r_2|r_1) + p(r_3|r_1) + 2p(r_2|r_1)p(r_3|r_1)] \\ &\quad + p(\bar{r}_1)[p(r_2|\bar{r}_1) + p(r_3|\bar{r}_1) + p(r_2|\bar{r}_1)p(r_3|\bar{r}_1)] \\ &= \beta_{r_1}[0.3 + 0.1 + 2(0.3)(0.1)] \\ &\quad + (1 - \beta_{r_1})[0.2 + 0.4 + (0.2)(0.4)] \end{aligned}$$

$$\begin{aligned} C(s_2) &= p(r_1) + p(r_2) + p(r_3) \\ &= p(r_1)[1 + p(r_2|r_1) + p(r_3|r_1)] \\ &\quad + p(\bar{r}_1)[p(r_2|\bar{r}_1) + p(r_3|\bar{r}_1)] \\ &= \beta_{r_1}[1 + 0.3 + 0.1] + (1 - \beta_{r_1})[0.2 + 0.4] \\ \therefore \beta_{r_1} &\approx 0.08 \end{aligned}$$

For this transition, we denote $\beta_{r_1} = \beta_1$; similarly, for the trigger on schedule s_1 to schedule $s_3 = 001$, the threshold β_2 can be set. The threshold setting for this example is illustrated in Figure 5. The cost of each schedule is a linear function of the probability of the root variable r_1. This can be validated by the equations described above. At the current state of operation $p(r_1) = 0.4$, and the best schedule is s_1. When $p(r_1)$ decreases below $\beta_{r_1} = 0.08$, the schedule $s_2 = 111$

becomes the optimal. Similarly, when $p(r_1)$ increases above $\beta_{r_2} = 0.97$, the best schedule changes to $s_3 = 001$.

Algorithm 2 Dynamic Ada($S, p, thresholds, iter$)

```
S := current schedule;
p := current probability vector of the k roots;
(β1, β2) := bounding threshold vectors on k roots;
iter := number of iterations for Ada to run;
begin procedure
status := β1 < p & p < β2; // k logical values
if status is not true then
  while iter > 0 do
    // until Ada runs into a local optimal
    while S' is not null do
      S':=changeSchedule(S', p); // Greedy search
      I:=S';
    end while
    record optimal over all iterations G:=min(I, G);
    S': a new seed schedule;
    iter:= iter-1;
  end while
  (β1, β2):= compute new thresholds on (G, p);
  status := β1 < p & p < β2;
  S := G;
end if
end procedure
```

Algorithm 3 changeSchedule(S, p)

```
begin procedure
mincost := scheduleCost(S, p);
initialize MIN to null;
for i = 1 : n do
  S':=Toggle i^th bit of S;
  C(S') = scheduleCost(S', p);
  if C(S') < mincost then
    mincost := C(S');
    MIN := S';
  end if
end for
return MIN
end procedure
```

Threshold setting employing such analytical equations can be extended to multiple roots. In the case of multiple roots, Ada assumes that the probabilities of the remaining roots remain unchanged while setting the threshold at a single root. Note that irrespective of the depth of the tree, the probability of any term $p(r_i, ..., r_j)$ will be linear in the root prior $p(r_1)$, since the conditional probability tables remain unchanged. Hence, the total cost of a schedule, and consequently the derived analytical equations for thresholds will always be linear functions of root probabilities and therefore easy to solve.

The complete technique to search for optimum is given in Algorithms 2 and 3. In Algorithm 2, we use an array of logical values $status$ in order to detect threshold violations in the current state S. When any of the thresholds is violated, $status$ changes; therefore, we need to re-optimize and run the Ada algorithm. In every iteration, the $changeSchedule$ function (in Algorithm 3) is invoked to realize a better cost. This algorithm is iterated until it reaches a local optimum, in which all neighbors impose a larger communication cost. The Ada algorithm is iterated over different seed schedules and the optimal over all the iterations is output.

5 Experimental Evaluation

In this section, we present a performance study demonstrating the features of our system. We used synthetic and real data in our experiments. The synthetic data set generates streams using b-model data generation that will be explained below. The real data contains an hour's worth of all wide-area traffic between the Lawrence Berkeley Laboratory and the rest of the world. The trace captured 1.3 million TCP packets on the Ethernet DMZ network, dropping about 0.0007 of the total [19]. We first describe our simulation environment.

5.1 Simulation environment

In order to study our techniques empirically, we built a discrete event simulator consisting of multiple data streams. We scheduled periodic tasks to initiate data arrivals for each stream. Measurements were collected on machines with dual AMD Athlon MP 1600+ processors, 2 GB of RAM, and running Linux 2.4.19. In our experiments, we varied four different system properties: the data characteristics, the network load, the memory budget, and the number of events monitored.

5.2 Synthetic data generation

Network traffic tends to be bursty. For example, we may observe a network router losing packets in bursts that are sudden and short lived. In order to model bursty nature of most real world data, Wang et al. [22] proposed an algorithm for generating a bursty time series of length $L = 2^k$ in space $O(\log L)$ and in time $O(L)$. The time series consists of Y data points, which is the number of total packets that arrives at the subnetwork.

5.2.1 Bursty time-series

The technique starts with an appropriate value for the parameter b, $0.5 \leq b \leq 1$, which determines the irregularity in the data. For example, $b = 0.5$ means uniformity, and $b = 1$ means extreme irregularity. Out of all Y data points, $Y * b$ of them are assigned to the first half of the series, and the remaining $Y * (1 - b)$ data points are assigned to the second half. This process continues recursively until L time points are generated.

5.2.2 Data characteristics

Routers that are at widely dispersed locations generate possibly independent events. In order to capture this environment characteristic, we used randomization in data generation for determining which half of the series gets the most number of data points. In case of a security attack, routers in a specific

region may start generating correlated events, i.e., there may be a temporal correlation between the packet arrival rates. We model such attacks in the network by injecting a randomly chosen subset of pairwise correlations in the data. Assume that we decide to incur a pairwise correlation between router N_i and router N_j. We generate a time series for router N_i using the above bursty model. We generate router N_j's data using router N_i's values as seeds.

We also generated another dataset in which the correlations were captured using first-order dependence trees. The whole network was broken down into multiple disjoint trees. In each tree, an internal node r_i with parent r_j samples the conditional probability values $p(r_i|r_j)$ and $p(r_i|\bar{r}_j)$ from a uniform distribution. Correlated binary data streams were generated using the above dependence trees. Experiments were conducted with the height of each tree ranging between 2 and 5, and the total number of nodes ranging between 10 and 40. In the dynamic case, updates to the priors at the root of each tree were sampled from a uniform random distribution.

5.3 Network setup

The largest number of nodes n we use is 100 for the case of independent streams, and 40 for correlated streams. As we pointed out earlier, different networks might have different traffic loads. Therefore, we experiment with a range of cost parameters to study the effect of network load. Table 2 shows the values of the parameters C_1 and C_2. Unless otherwise stated, we use $C_2 = C_3$.

C_1	10^{-3}	10^{-2}	10^{-1}	1	1	1	1
C_2	1	1	1	1	10^{-1}	10^{-2}	10^{-3}
$ratio$	10^{-3}	10^{-2}	10^{-1}	1	10	10^2	10^3

TABLE 2
Push/Pull ratios used in the experiments.

5.4 Competing Techniques

In our experiments, we measure the communication cost, which is the sum of the push cost and the poll cost. The poll cost consists of the pull cost and the ensuing answer cost. We measure the computational complexity in terms of execution time. We compared our algorithms *SPA* and *Ada* with three other techniques:

1) *all-push* scheme: if a router detects an event, it immediately reports the event to the monitoring station.
2) *all-pull* scheme: routers wait for explicit pull requests from the monitoring station.
3) *improved-value*: a value based monitoring algorithm by Raz et al. [7]. We explain the algorithm in Section 5.6.1.

5.5 Adaptivity to network conditions

In order to demonstrate the effect of network load on optimization, we compare the performance of the SPA and Ada with the other two non-adaptive algorithms, i.e., with all-push and all-pull.

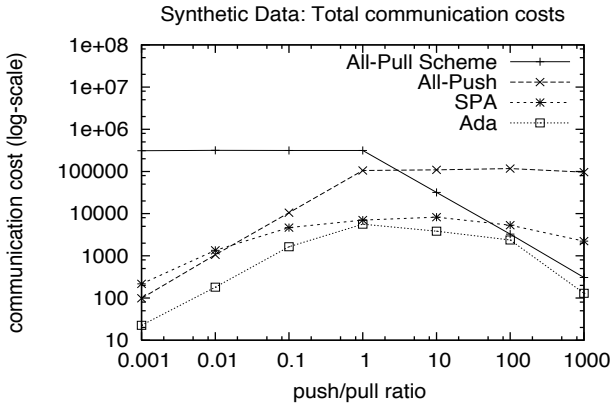


Fig. 6. Communication cost with varying push/pull ratio on a network of $n = 40$ events.

This experiment was performed on the synthetic data set with $n = 40$ events. Figure 6 shows the results. Our algorithms, especially Ada, adapts to the changing network conditions by favoring push over pull for small ratios, and vice versa for large ratios. This adaptivity in responding to changes in the network is a key benefit of our algorithms, since they can be applied over any range of network traffic conditions.

5.6 Event Independence

In this section, we assume that the query registered at the monitoring station involves all n events which are independent of each other.

5.6.1 Case study: monitoring network traffic

In this section, we look at detecting an alarm condition on n real-valued variables $x_1, \ldots, x_n$. Let x_i^t denote the value of interface out traffic x_i at node N_i at time t. Our goal is to detect when $\sum_{i=1}^{n} x_i^t > T$. The naive solution, i.e., all-push, for this problem is as follows: at time t, if the current local value x_i^t exceeds the budget T/n, then send the value x_i^t to the station. At time t, if the station received one or more event reports, then it polls all other nodes for their current values. If the total sum exceeds T, then we generate an alarm. This simple algorithm has a clear disadvantage when n is big, since its expected cost ratio compared to all-pull (global poll) goes to one [7]. The authors improve this simple value-based algorithm by reducing the "budget" given to each local node, in a way that a single event driven report will not force the initiation of a global poll. For this purpose, they assume an upper bound D (hypothetically the interface speed) on the value of x_i, and they issue a global poll only when l ($1 \leq l \leq n$) or more local variables exceed a threshold τ, which is smaller than T/n: to ensure correctness we should have $(l-1) * D + (n - (l-1)) * \tau \leq T$, which implies

$$\tau \leq \frac{T - (l-1) * D}{n - (l-1)} \tag{2}$$

Note that for $l = 1$, this reduces to the naive solution above. Our key observation here is that our optimizer can schedule l

nodes to push. The remaining $n-l$ nodes are scheduled to be conditionally pulled, which happens when there is a possibility of an alarm condition: let r_i^t denote the binomial event $x_i^t \geq \tau$. If $\forall\ r_i \in \mathcal{R}^+$such that $r_i^t = 1$, then we issue a global poll. Our optimal algorithm mentioned in Section 3.1 identifies the top-l least probable event set and schedules them for push.

The authors assumed the communication cost of polling is the same as the communication cost of pushing. Therefore, we set $C_1 = 1$, $C_2 = 1$ and $C_3 = 0$. We use WAN-traffic data averaged with a moving window of size 256. We partition the data into 50 small streams of size 10000, one for each of $n = 50$ nodes (events). We set the global threshold T to $0.8nD$, where n is the number of nodes and D is the maximum value achievable. In this experimental setting D was equal to 1.003×10^4.

Figure 7 shows the results. Our algorithm which monitors top-l least frequent events, incurs minimal communication cost for $1 \leq l \leq 40$. Each node in our scheme pushes its probability value aperiodically to the station, if the current probability value deviates by more than a slack of 0.1 from the last reported value. Even though *improved-value* issues a global poll only when l nodes exceed τ, it is highly likely to find a set of l high-frequency events that fire at the same time. Since our algorithm monitors the l low-frequency events, there is a very low likelihood of a global poll. This results in substantial cost savings.

As the value l increases, the threshold τ decreases and hence, the probability of occurrence of local events increases. This leads to higher communication costs for both the schemes. When l reaches 41 the threshold drops to 0 since $((l-1)D = 40 * D = 0.8 * n * D = T)$), and every event starts firing. At this point, both the schemes push 41 events and poll the remaining 9 events. When l reaches 45, both the schemes push 45 events and poll the remaining 5 events. Since $C_1 = C_2 = 1$, and $C_3 = 0$, the cost of pushing and pulling are equal. Hence, the total cost attains the maximum and remains a constant after $l = 41$.

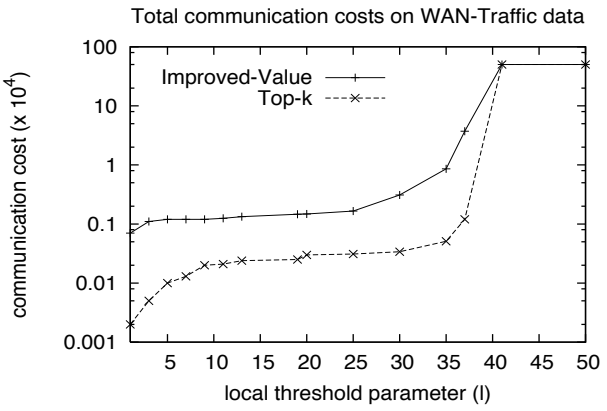


Fig. 7. Communication cost with varying threshold parameter l on WAN-traffic ($w = 256$) for $n = 50$.

Figure 8 shows the communication costs of the scheme with varying slack parameter δ on monitoring event probabilities. Updates to the probability values at each node are sent to the Base Station, only if they exceed the slack value δ. The test configuration is identical to the previous experiment with l set to 30. The total cost of the improved value scheme is independent of the slack term and hence is a constant. On the other hand, the total cost of the top-k scheme composes of two terms: a)the cost of monitoring the event probabilities and b) the incurred push/pull costs. The first term, i.e., the cost involved in monitoring the probabilities is inversely proportional to the slack value. Hence, at slack values smaller than 0.1 the monitoring costs are quite high and offset the optimization benefits of the top-k scheme; whereas, at large slack values the monitoring costs are quite low, but the performance of the top-k scheme is also lowered, as the optimizer operates on stale event probabilities.

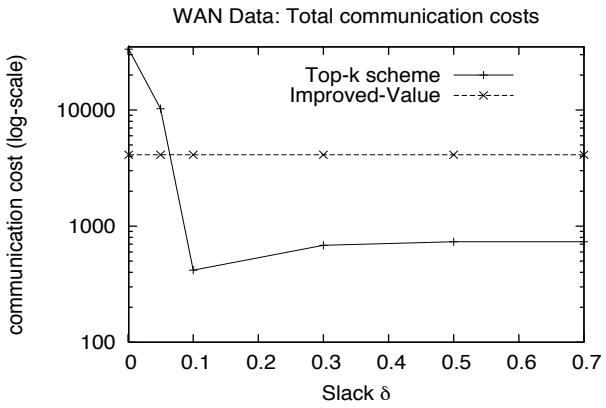


Fig. 8. Total Communication cost with varying slack parameter δ on WAN-traffic ($w = 256$) for $n = 50$.

5.6.2 Scalability with number of events

We use WAN-traffic data with a moving average window of size 64. We compare the performance of our algorithm with the all-push and the improved value schemes in the cost configuration $C_1 = 1$, $C_2 = 1$ and $C_3 = 1$. The threshold τ is calculated using Equation 2. For the All-Push scheme, any local value that exceeds τ is caught by local traps. We measure the scalability of our optimizer with the number of events. Figure 9 shows the results. All-push incurs a monitoring cost that is up to two orders of magnitude more than our optimizer. The performance of our technique is commendable and is the result of considering the low frequency events. However, this improvement comes at the cost of computational overhead at the station. Figure 10 shows the *total* time the algorithms take to run to completion. Even though our total computational costs are higher than the competing schemes, the average computational cost for optimization, as seen later, is within the tolerable limit.

5.7 Conditional Dependence

In this section, we first consider the scalability of our algorithms with varying number of events and then measure the memory and computational overheads of each algorithm.

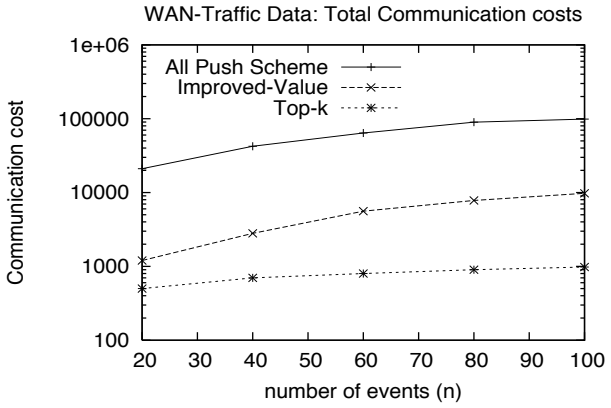


Fig. 9. Communication cost with varying number of events for monitoring an all-events query on WAN-traffic data.

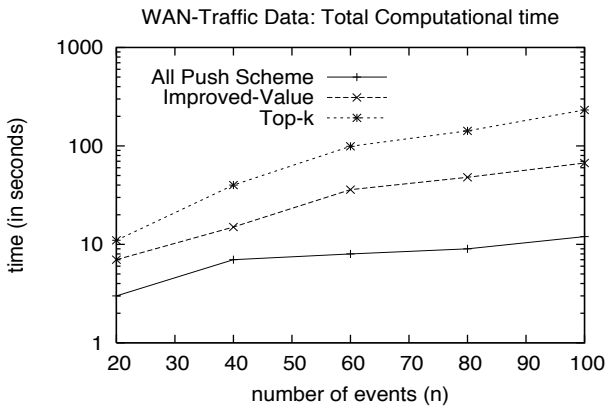


Fig. 10. Computational cost with varying number of events for monitoring an all-events query on WAN-traffic data.

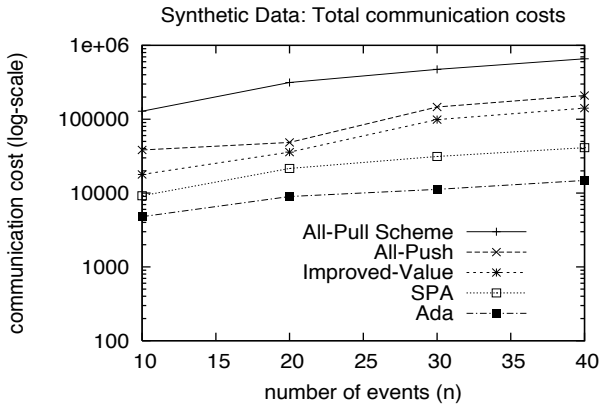


Fig. 11. Communication cost with varying number of events on synthetic data.

5.7.1 Scalability with events

Figure 11 plots the communication costs of the SPA and Ada algorithms with the competing techniques. The plot shows that our algorithms outperform the competing algorithms by almost two-orders of magnitude. The SPA algorithm was alloted a memory budget B of size $400KB$. Since $C_1 = C_2 = C_3 = 1$, the cost of the All-pull scheme is always higher than the cost of the All-push scheme. As the SPA algorithm evaluates the minimum cost schedule over a large set of samples (10^4) it incurs a smaller communication cost compared to the other two techniques. Similarly, Ada is quite successful in executing a greedy search for the optimal. It either ascertains the optimal schedules for small event sizes or schedules with a very small cost for large event sizes.

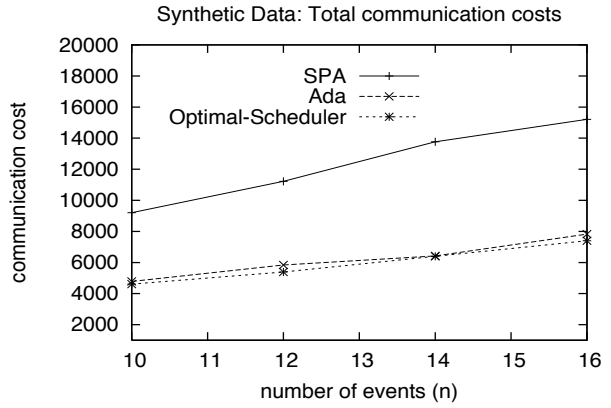


Fig. 12. Comparison with the optimal scheduler on the synthetic data.

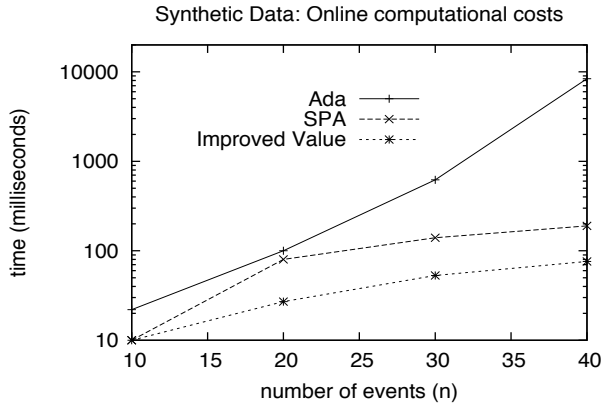


Fig. 13. Average computational cost with varying number of events on synthetic data.

Figure 12 compares the communication costs of the SPA and Ada algorithms with a static optimizer which generates the optimal schedules. The optimizer enumerates schedules to calculate the minimum cost schedule. The figure depicts that Ada's performance is (almost) optimal in most of the cases. This experiment could be performed only at small event sizes because i) evaluating the cost of all the schedules is a computationally expensive procedure and ii) recalculating the optimal for each update to the event distribution makes it infeasible to employ the static optimizer.

Figure 13 illustrates the average computational response times of all the schemes. Evaluating the cost of a schedule took around 3 milliseconds (on the average) for event size $n = 10$ and around 30 milliseconds for $n = 40$. The SPA

algorithm bypasses the expensive computation of online cost computation by a) reusing the partial costs evaluated a priori to compute the total cost and b) employing the threshold-based pruning on the sorted set of lists to evaluate a small set of candidates.

For small event sizes, we observed that the threshold-settings of Ada result in large savings. Further, even if the thresholds are violated, the number of iterations to reach the optimal were small. Hence up to a network size of $n = 30$ Ada provides good response times of 0.5 seconds. However, for large event sizes, Ada has to iterate over a large seed size (of up to 100) in order to find a schedule with a small cost as the search space explodes exponentially in n. As a result, we noticed that Ada has high computational costs.

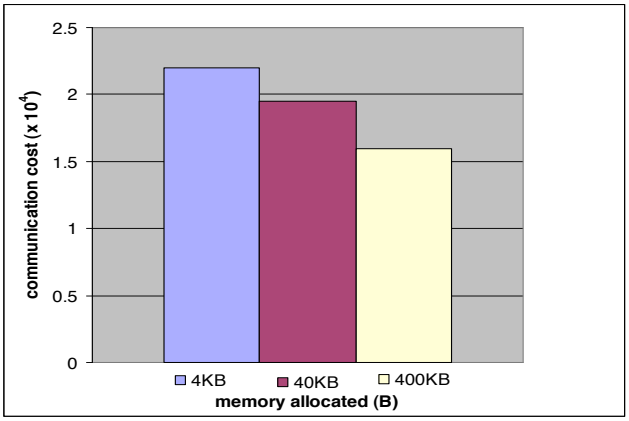


Fig. 14. SPA algorithm: Communication cost with varying memory budget size on synthetic data.

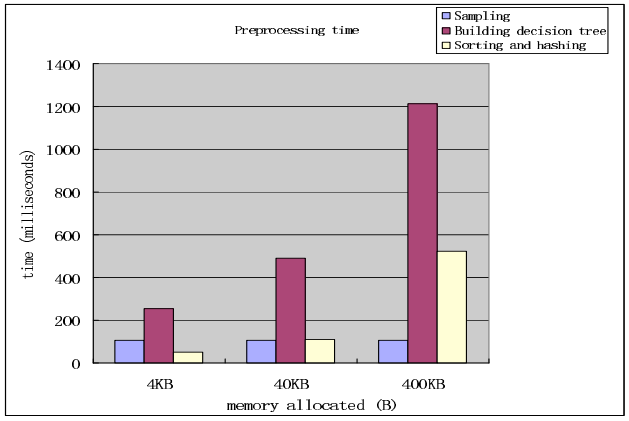


Fig. 15. Preprocessing cost with varying memory budget for the SPA algorithm on the synthetic-data set.

5.7.2 Memory and Computational Overhead

Figure 14 plots the communication costs of the SPA algorithm with varying memory budget (B) for $n = 20$. When the memory budget increases, the SPA algorithm searches for the optimal over a larger space of regular expression classes and hence computes schedules with smaller cost. Figure 15 illustrates the preprocessing overhead involved in maintaining different memory budgets for a fixed sample size of 10^4 samples. We have decomposed the computational cost into the different components involved in sampling, building the decision tree, sorting the lists and maintaining a hash table for providing a random access into each list. The predominant cost as seen from the figure lies in building the decision tree, since the *GreedySplit* algorithm is invoked at each step to classify the samples into D classes. For all the memory sizes, we have omitted the computational time involved in evaluating the schedule cost over the large set of samples. This cost evaluation is independent of B and was measured to be approximately 210.23 seconds for this sample set.

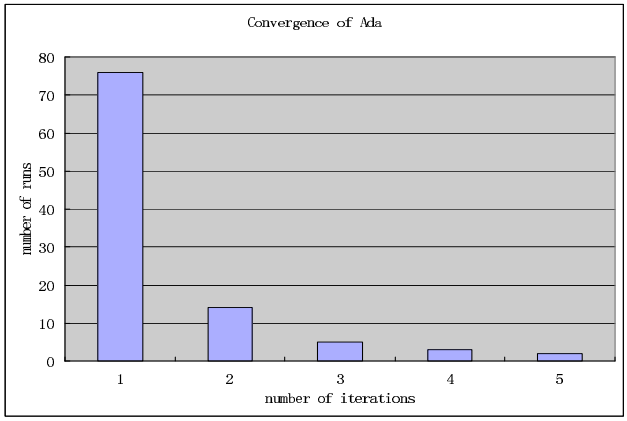


Fig. 16. Convergence of Ada to the optimal cost in number of iterations, for $n = 10$ events.

5.7.3 Convergence of Ada

Figure 16 depicts the convergence rate of Ada. The figure plots the number of iterations required by Ada in order to converge to the optimal schedule for a network of size $n = 10$. The optimal solution was determined by enumerating the costs of all schedules. The experiment was performed for 100 runs. The histogram plot shows that in 90% of the runs, Ada converged to the optimal within 2 iterations.

5.8 Summary of experimental results

We summarize our experimental results and the corresponding tradeoffs as follows:

- In the case of independent streams, our optimal algorithm which monitors the top-k least frequent event set outperforms the competing schemes by up to two-orders of magnitude.
- In the case of correlated streams, our algorithms *Ada* and *SPA* outperform the all-push, improved-value and all-pull schemes; further, they adapt well to changing network conditions.
- If communication cost is the *only* over-riding concern, then Ada algorithm is the best scheme.
- If computational response times of less than 0.5 seconds are required, then for network sizes of up to 30 nodes, Ada is preferred over SPA. Beyond 30 nodes, the SPA algorithm is preferred.

6 Related work

Olston et al. [1], [18] show how to trade precision for performance using approximations in a distributed system. A central station coordinates a number of distributed sites, and installs individual constraints in each site. These constraints specify the amount of deviation that a site value can have from its last reported value to the station without violating the query invariants. As long as the invariants hold at each site, no message communication is necessary. This work is complementary to our work, where we need to identify the top-k least frequent event set continuously. Chu et al. [4] extend the above work to a sensor network setting, and exploit the spatial-correlations among the attributes to further reduce communication costs. This work can be considered to be optimization over strategies which employ only the push mechanism. Jain et al. [12] consider the resource management problem as a filtering problem, where the objective is to filter out as much data as possible to conserve system resources, and at the same time to meet users' precision requirements. For this purpose, they employ a dual Kalman filter approach where a filter at the central server mimics the filter at the remote source in order to predict the source values for conserving communication resources.

Deshpande et al. [6] propose a new framework for identifying correlations among attributes in order to reduce communication cost in acquisitional query processing. When executing queries with several predicates over attributes with high acquisition costs, it is often times beneficial to re-write the query by introducing correlated attributes with relatively lower acquisition cost in order to filter out non-potential candidates early in the query stage with minimal cost. Their cost model is similar to our cost model in this paper, except that they focus on producing a sequential-pull query plan which determines greedily the next attribute to pull given the set of already pulled attributes. Unlike our paper, they do not consider the push strategy during optimization.

Zhu et al. [23] consider techniques for disseminating erratic data streams stored in a server to interested clients. Erratic data such as sensor streams, stock prices, and more importantly network statistics, change frequently and unpredictably. Therefore, a linear change model does not capture the source data characteristics adequately. Instead, a Brownian motion model (non-linear change model) is shown to achieve a higher fidelity on erratic data, compared to other simpler change models as in the above case. In our monitoring system, rather than caching and consistency issues due to local erratic data changes, we are mainly interested in those times when the current data profile over a relatively short time period do not conform to the long running base profiles over larger time periods. Furthermore, the communication in our model is between the erratic data source and the server; therefore, an adaptive push-pull model is more adequate in our settings.

More recently, Kifer et al. [14] proposed a change-detection model in a streaming context, where base-line normal activity profiles are compared against running activity profiles. Non-parametric statistics are computed and used in thresholding for alarms. Each network element in our monitoring system can use this scheme to decide when to push, and reduce communication with the server considerably. This approach corresponds to modeling the stream, and transmitting the model itself. The model gets updated in case of a "major" shift in data distribution.

Dilman and Raz [7] proposed the original reactive network monitoring problem of achieving significant reduction in management overhead by combining global polling with local event driven reporting. Their solution does not exploit the correlations across events nor the frequency of occurrence of events. By modeling the correlations across events as a Bayesian Network, and designing algorithms which exploit such dependencies, we further reduce communication costs.

Massie et al. [17] consider a distributed and hierarchical system "Ganglia" that monitors a large number of clusters. Each cluster is maintained as a single entity, and all nodes within the same cluster always have an approximate view of the entire cluster state. This approach is feasible under frequent failures, which is typical of clusters. Meta nodes at higher levels of the hierarchy federate multiple clusters using point-to-point connections with representative nodes of its children clusters. The monitoring system is currently being used in clusters, Grids, and planetary-scale systems. We can deploy our framework on top of Ganglia, which will provide a fault tolerant and resilient communication medium.

Ramamritham et al. [5], [21] consider adaptive data dissemination techniques for dynamic data. In their approach, users specify individual coherence requirements over data values, and the system tries to guarantee that users' view of the data is never out-of-sync by more than their coherence requirements. The scheme applies a linear change model on source objects. The degree to which coherence requirements are met defines the system's *fidelity*. There are two different ways of communication in the system: "servers" can push the data to "proxies", which in turn can push the data to interested "clients". Or, proxies can pull the data from the servers. Given the resource constraints at the server and the available bandwidth, the system adaptively chooses between push and pull for each existing data connection to increase the fidelity in the face of data source crashes. Our work is different than this model, since we look at optimizing the schedule of pushes and pulls from a number of data sources given an arbitrary query workload, rather than locally optimizing each data connection.

7 Conclusions

In this paper, we presented a framework for monitoring global system parameters as a function of local properties of network elements. We considered the scheduling of network elements for a given probability of occurrence of events such that the monitoring cost in terms of message exchanges is minimal. We designed an optimal algorithm when the events are independent and proved that optimal solution is NP-complete when the events are conditionally dependent. We proposed two efficient solutions, the SPA and the Ada algorithms which employ greedy techniques in the search for schedules with low costs. The SPA technique precomputes partial costs, and

the Ada algorithm employs a threshold-setting scheme in order to decrease the reoptimization cost as the environment characteristics change over time. Our extensive evaluation on both the real-world and synthetic data sets show that our algorithms outperform the competing techniques by two orders of magnitude.

8 Future Work

There are several avenues for future work: if we consider dependencies regarding pull events by employing a sequential pull strategy, we can decrease the total cost of a schedule. However in this case, the probability formula is more complex than just a product term. Furthermore, the tradeoff between the accuracy and the computational cost in case of using higher order components for approximating multivariate distributions is an open issue to explore. Currently, the paper considers the conjunctive query (an *and* operator) only; however, the problem becomes more interesting and complex if we introduce the disjunction (the *or* operator). Cost optimization over a predicate involving both the operators, expressed in a CNF or DNF [9] form is an interesting direction of research.

Extending Ada's analytical thresholding scheme to the setting where the priors at each tree are allowed to change simultaneously is an interesting direction of our future research. Since the priors evolve at a slow rate, we can assume that all updates to the priors are bounded within a slack of δ. If we remove the slack assumption, then the analytical equations are transformed to polynomials of order k, where k is the number of roots. This corresponds to setting the threshold in the k-dimensional space of root priors.

References

[1] B. Babcock and C. Olston. Distributed Top-K monitoring. In *SIGMOD*, pages 28–39, 2003.

[2] A. Bulut, A. K. Singh, N. Koudas, and D. Srivastava. Adaptive reactive network monitoring. Technical report, UC Santa Barbara, March 2005, http://www.cs.ucsb.edu/~bulut/bsks05tech.pdf.

[3] C. Chow and C. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Trans. on Information Theory*, 14(3):462–467, 1968.

[4] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong. Approximate data collection in sensor networks using probabilistic models. In *ICDE*, pages 48–60, 2006.

[5] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: disseminating dynamic web data. In *WWW*, pages 265–274, 2001.

[6] A. Desphande, C. Guestrin, S. Madden, and W. Hong. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, page to appear, 2005.

[7] M. Dilman and D. Raz. Efficient reactive monitoring. In *INFOCOM*, pages 1012–1019, 2001.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, pages 102–113, 2001.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, 1 edition, 1979.

[10] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *SIGMOD*, pages 13–24, 2001.

[11] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2 edition, 2006.

[12] A. Jain, E. Chang, and Y. F. Wang. Adaptive stream resource management using kalman filters. In *SIGMOD*, pages 11–22, 2004.

[13] J. Jiao, S. Naqvi, D. Raz, and B. Sugla. Toward efficient monitoring. *IEEE Journal on Selected Areas in Communications*, 18(5):723–732, May 2000.

[14] D. Kifer, S. Ben-David, and J. Gehrke. Detecting change in data streams. In *VLDB*, pages 180–191, 2004.

[15] J. Kleinberg. Bursty and hierarchical structure in streams. In *SIGKDD*, 2002.

[16] H. Ku and S. Kullback. Approximating discrete probability distributions. *IEEE Trans. on Information Theory*, 15(4):444–447, 1969.

[17] M. Massie, B. Chun, and D. Culler. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7), July 2004.

[18] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, pages 563–574, 2003.

[19] V. Paxson and S. Floyd. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[20] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, 1988.

[21] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *VLDB*, pages 57–68, 2003.

[22] M. Wang, T. Madhyastha, N. Chan, S. Papadimitriou, and C. Faloutsos. Data mining meets performance evaluation: fast algorithm for modeling bursty traffic. In *ICDE*, pages 507–516, 2002.

[23] S. Zhu and C. Ravishankar. Stochastic consistency, and scalable pull-based caching for erratic data sources. In *VLDB*, pages 192–203, 2004.

Appendix A
Approximating the probability distribution

A method for optimal approximation of an n-variate discrete probability distribution by a set of $n-1$ second order component distributions using first order dependence relationships was considered by Chow and Liu [3]. Out of a possible $\binom{n}{2}$ second-order component distributions, the authors approximate a multivariate probability $p(\mathbf{q})$ using at most $n-1$ of these lower order component distributions as

$$p_a(\mathbf{q}) = \prod_{i=1}^{n} p(r_{m_i} | r_{m_{j(i)}}),\ 0 \leq j(i) \leq i \tag{3}$$

where $m_1, \ldots, m_n$ is an unknown permutation of the integers $1, \ldots, n$. By definition, $p(r_i|r_0)$ is equal to $p(r_i)$. Each variable is conditioned on at most one other variable, and the dependence relationships can be represented by a tree called *first-order dependence tree*. If $j(i)$ is 0 for exactly one variable, the tree is connected and has $n-1$ branches. Otherwise, it is a forest of trees.

The goodness of approximation $p_a(\mathbf{q})$ is defined in terms of the discrimination information as

$$I(p, p_a) = \sum_{q} p(q)\ \log\ \frac{p(q)}{p_a(q)} \tag{4}$$

Minimizing this closeness measure $I(p, p_a)$ is equivalent to maximizing the total branch weight in the dependence tree representation [3]. The weight on a given tree edge between r_i and $r_{j(i)}$ expresses the mutual information between r_i and $r_{j(i)}$, which is equal to

$$I(r_i, r_{j(i)}) = \sum_{r_i, r_{j(i)}} p(r_i, r_{j(i)})\ \log\ \frac{p(r_i, r_{j(i)})}{p(r_i)p(r_{j(i)})} \tag{5}$$

where we use a maximum likelihood estimator computed on the available samples to approximate the joint probability.

The tree representation of maximum branch weight can be constructed using Kruskal's spanning tree algorithm after

we compute and sort all $\binom{n}{2}$ mutual information measures in descending order. The whole operation takes $O(n^2 \log n)$ time. Edges are selected in the given order, and tested for inclusion into the on-going tree representation. An edge is chosen to be in the representation if the addition of the edge does not create a cycle in the tree. We compute this tree periodically. Therefore, the computational cost is amortized.

In order to get better approximations, one can also use more than $n-1$ second order components or possibly higher order components (higher than two); however this comes at a higher computation cost: a convergent iterative procedure may need as many as $\binom{n}{i}$ i-th order components to obtain the optimal approximation. This amounts to maintaining $O(n^2)$ second order components for $i = 2$, and $O(n^3)$ third order components for $i = 3$, which also equals to the time and space complexity of the procedure. The details of this scheme can be found in [16]. Due to its overheads, we do not explore this scheme any further; we restrict ourselves to $n-1$ second order components. This has the effect of (a) keeping the state space small in size and (b) allows us to utilize algorithmic techniques (as opposed to iterative) to obtain the best approximation.

APPENDIX B
HARDNESS OF OPTIMIZATION IN THE CONDITIONAL DEPENDENCE CASE

In this section, we show that the optimization problem is intractable for the case of conditional dependence. Given a joint distribution represented by a first-order dependence (Bayesian) tree, the problem resolves to ascertaining the minimum cost schedule in the class χ_k, i.e., the class composing of schedules with k pushes. In fact, we prove a stronger result and show that problem is an instance of the 0-1 Integer Programming problem even when the first-order dependence tree degenerates to a forest of edges. Consider the formulation of the problem in 6, where $|\mathcal{R}^+| = k$.

$$\sum_{r_i \in \mathcal{R}^+} p(r_i) + p(\mathcal{R}^+) * \left(\sum_{r \in \mathcal{R}^-} C_2 + p(r|\mathcal{R}^+) C_3 \right)$$

Assume that the dependence tree is of the following form, where network element r_2 is dependent *only* on event r_1, and event r_4 is dependent only on event r_3, and so on. In the general case, r_{i+1} is dependent on r_i for all i's which are odd. Hence the probabilities $p(r_{i+1}|r_i)$ and $p(r_{i+1}|\bar{r}_i)$ are given by the conditional probability tables (known a priori) if i is odd. All other pairs of events are conditional independent.

Now, we can transform the above problem expressed above into an equivalent 0-1 Integer Programming formulation as follows. The Integer Programming formulation was shown to be NP-hard [9]. Assume that for each r_i, there is a corresponding variable y_i which is set to 1 if r_i is set to push and 0 otherwise. Then, Equation shown in 6 can be formulated as the following problem:

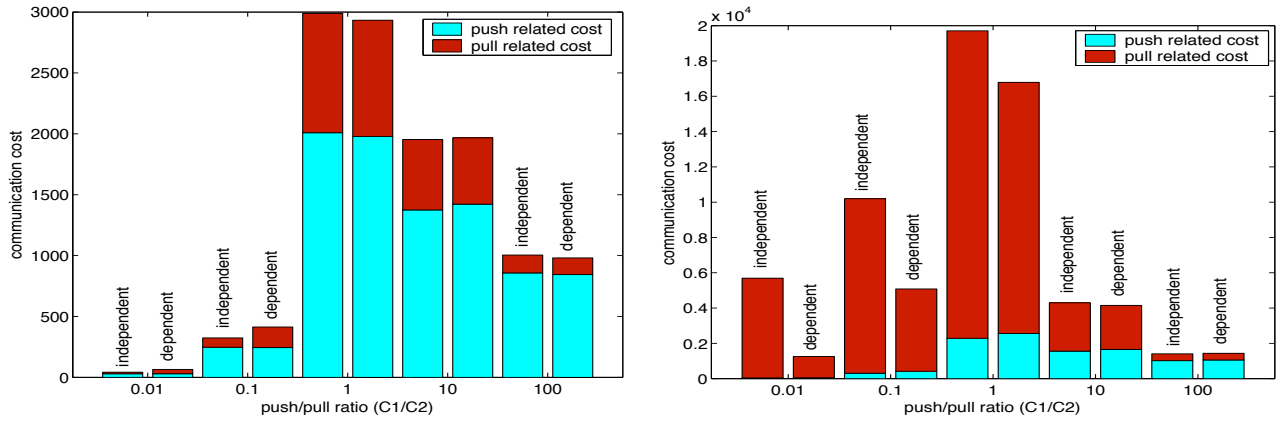
$$\begin{aligned}
&\text{Minimize} \quad C_1 * \mathcal{S}_1 + C_2 * \mathcal{S}_2 + C_3 * \mathcal{S}_3, \text{ where} \\
&\mathcal{S}_1 = \sum_{i=1}^{n} y_i * p(r_i) \quad \text{// push costs} \\
&\mathcal{S}_2 = \mathcal{J} * (n-k) \quad \text{// pull costs} \\
&\mathcal{S}_3 = \mathcal{J} * \sum_{all\ (r_j, r_{j+1})\ dependencies} \mathcal{F}_j \quad \text{// answer costs} \\
&\mathcal{J} = \prod_{all\ (r_j, r_{j+1})\ dependencies} \mathcal{D}_j \\
&\mathcal{D}_j = y_j y_{j+1} p(r_j, r_{j+1}) + y_j (1 - y_{j+1}) p(r_j) \\
&\qquad + (1 - y_j) y_{j+1} p(r_{j+1}) + (1 - y_j)(1 - y_{j+1}) \\
&\mathcal{F}_j = (1 - y_j)(1 - y_{j+1})[p(r_j) + p(r_{j+1})] + \\
&\quad = y_j (1 - y_{j+1}) p(r_{j+1}|r_j) + (1 - y_j) y_{j+1} p(r_j|r_{j+1}) \\
&\text{subject to constraints} \sum_{i=1}^{n} y_i = k
\end{aligned}$$

The above formulation splits up the total costs into push costs, the $\mathcal{S}_1$ term, the costs involving the conditional pull requests, the $\mathcal{S}_2$ term, and the costs involving the answers to the requests, the $\mathcal{S}_3$ term. $\mathcal{J}$ denotes the probability of pull, i.e., $p(\mathcal{R}^+)$. Each term in $\mathcal{J}$, $\mathcal{D}_j$ denotes the contribution of events r_j and r_{j+1} to $p(\mathcal{R}^+)$ under all possible combinations: 1) if both the events are present in the push set 2) if only one of them is present in the push set or 3) if none of them is present in the push set. Each term $\mathcal{F}_j$ denotes the conditional probability of answering the query. It can also decomposed into the three cases as above.

APPENDIX C
EXPERIMENTS CAPTURING DATA CHARACTERISTICS

Finally, we study the effect of data characteristics on performance of *Ada* while varying push/pull ratio. We use independent and correlated streams that are generated synthetically (see Section 5.2). Figures 17(a) and 17(b) show the results for independent streams and correlated streams respectively. For each ratio value, the first column shows the result for *Ada* assuming independent events and the second for *Ada* assuming joint-dependence on events. Since push cost is modeled in the same manner for both schemes, the pull cost will be important in differentiating the overall performance. First, consider the case of independent streams. Answers with regards to pull requests are not likely when the underlying data streams generate independent events. This implies that the pull cost will be small compared to the push cost. Therefore, we expect both schemes to perform similarly. As shown in Figure 17(a), the overall performance of both schemes are comparable for all ratios.

However, when we consider correlated streams, the pull cost starts to dominate due to the increased likelihood of co-occurrence. For small ratios, the pull requests and answers have a larger cost compared to the push messages. Therefore,



(a) Independent streams

(b) Correlated streams

Fig. 17. Effect of data characteristics on performance of *Ada* on a network of $n = 15$ events. We measure communication cost vs. varying push/pull ratio. For each ratio, the first column shows the result of *Ada* assuming independent events and the second for *Ada* assuming dependence on events.

Ada (dependent) scheme outperforms its counterpart by up to five times as shown in Figure 17(b). For large ratios, the push messages are far more expensive than the pull requests and answers, and drive the trend in the overall cost. Therefore, the performance of both schemes are similar.

The decrease in the total cost for the ratios 1, 10 and 100 in case of both independent and correlated streams is due to the decreasing number of events that are pushed.

APPENDIX D
MULTIPLE QUERIES

In this section, we extend the optimization problem of monitoring a single query to multiple queries. The problem formulation allows for cost reduction across multiple queries by allowing the sharing of push events across multiple queries.

Let $\mathcal{R}$ denote the set of all local events. We use $\mathcal{R}^+$ to denote the set of events that are pushed, and $\mathcal{R}^-$ to denote the set of events that are pulled in a given schedule S. Let Q with cardinality m denote the set of queries registered at the monitoring station. For each query $Q_j \in Q$, where $1 \leq j \leq m$, we define two sets:

$$\begin{aligned} Q_j^+ &= \{r_i | r_i \in Q_j \cap \mathcal{R}^+\} \\ Q_j^- &= \{r_i | r_i \in Q_j \cap \mathcal{R}^-\} \end{aligned}$$

Each event r in Q_j^- is pulled only if all events in Q_j^+ occur, which happens with probability $p(\cap_{r_i \in Q_j^+} r_i)$=$p(Q_j^+)$. The answer to this pull request occurs with probability $p(r|Q_j^+)$, which is the conditional probability of occurrence for r given that all push events occurred. For example, let the schedule S be 010. Then, the sets for query Q_1 are $Q_1^+ = \{r_2\}$ and $Q_1^- = \{r_1, r_3\}$, and for query Q_2 are $Q_2^+ = \{r_2\}$ and $Q_2^- = \{r_1\}$. We assume *no sharing* of pull events between queries at the station. We can formulate the cost $C(S)$ of S in terms of the probabilities $p(r_i)$'s, and the cost parameters C_1, C_2, and C_3 as follows:

$$= \overbrace{p(r_2)C_1}^{push} + \overbrace{p(r_2)(C_2 + p(r_1|r_2)C_3)}^{pulling\ for\ Q_2^-} + \overbrace{p(r_2)(C_2 + p(r_1|r_2)C_3) + p(r_2)(C_2 + p(r_3|r_2)C_3)}^{pulling\ for\ Q_1^-}$$

where each individual term denotes the expected cost.

D.0.1 Scalability with number of queries

Ada has a linear time dependency in the number of queries registered at the station. In order to validate our analytical expectation, we measured the execution time while increasing the number of queries m. Our experiments were run on a test network of size $n = 40$, with unit message costs and for various query workloads. Table 3 summarizes the results we obtained and confirmed that the time dependency is in fact linear in terms of the number of queries m.

m	1	5	10	15	20	25	30	35
time	20	49	82	112	155	198	235	263

TABLE 3
Execution time of Ada (in seconds) with varying number of queries m on $n = 40$ events.

APPENDIX E
ADA'S REACHABILITY OF OPTIMAL STATE

A monitoring algorithm that decides which variables to measure based up on values obtained in the past is *optimal*, if there is no other correct algorithm with less cost. Jiao et al. shows that optimal monitoring algorithms may not exist [13]. One can create scenarios where the neighborhood of the current operating state, which was previously optimal, and is currently

a suboptimal state, does not allow any transitions, because each neighbor state imposes a larger communication cost. If the number of variables is large, reaching a global optimum would require a random jump of some sort and we show that this jump is more than a constant number of edges:

Theorem E.1: Let S_0 denote the optimal solution for some initial configuration. We show that there exists an execution scenario in which

- There will be a T-step neighbor S_{opt} of S_0, and it will have the smallest cost.
- All the other K-step neighbors of S_0, where $1 \leq K \leq T$, will have larger costs than S_{opt}.

A T-step neighbor being the global optimum for an arbitrary T implies that a very large neighborhood needs to be searched for finding the global optimum.

Proof: The proof follows from a series of lemmas which we will show next.

We will show the proof in the generalized setting of multiple queries as defined above. Assume that we have T queries such that each query Q_i for $3 \leq i \leq T+2$ is expressed as $Q_i : \{r_1, r_2, r_i\}$. Therefore, the ranges r_1 and r_2 are the only common elements among the queries. Let each $p(r_i)$ be equal to δ where $0 \leq \delta \leq 1$. In the ensuing development, we will assume that the events are independent, and the message costs are $C_1 = C_2 = C_3 = 1$.

For a specific set of initial values of δ and T, the optimal schedule is $S_0 = 1100\ldots0$, which pushes from r_1 and r_2, and pulls from all the other ranges. With all other system variables constant, we will keep increasing $p(r_2)$, and prove that at some point in time for a specific value of $p(r_2)$, the optimal schedule will be $S_{opt} = 1011\ldots1$, which pushes from all ranges except r_2. Furthermore, S_{opt} cannot be reached from S_0 since its immediate neighbors will have larger costs than itself. For reaching global optimum S_{opt}, a random jump that involves T+1 steps is required. This also proves that a very large neighborhood needs to be searched to find the global optimum, since T is arbitrary.

In order to prove our claim, we will consider the schedules that start with 11, 01, or 10. Each of these schedules are followed by any combination of 0s and 1s for the remaining T ranges. There is only one viable schedule that starts with 00, which is followed by T 1s, since we need to have at least one push for each query. We enumerate the communication cost for the set of schedules as follows: we use $(i) = cost(11\ldots)$, $(ii) = cost(01\ldots)$, $(iii) = cost(10\ldots)$ and $(iv) = cost(001\ldots1)$,

$$
\begin{aligned}
&i) \quad (T-K)\delta p(r_2)(1+\delta) + K\delta + p(r_2) + \delta \\
&ii) \quad 2(T-K)p(r_2)(1+\delta) + p(r_2) + K\delta + K\delta p(r_2)(1+\delta) \\
&iii) \quad (T-K)\delta(1+p(r_2)) + (T-K)\delta(1+\delta) + \\
&\qquad K\delta^2(1+p(r_2)) + K\delta + \delta \\
&iv) \quad T\delta(2+\delta+p(r_2)) + T\delta
\end{aligned}
$$

where $K \leq T$ denotes the number of pushes among the ranges r_i for $3 \leq i \leq T+2$. The schedule S_0 is in set (i) for $K = 0$. Immediate neighbors of S_0 are the schedules for $K = 1$ in (i), the schedule for $K = 0$ in (ii), and the schedule for $K = 0$ in (iii). We will proceed step by step to prove our claim. We start off by showing that:

Lemma E.1: For all schedules in (i), the cost *increases* with *increasing* K.

Proof:

$$
\begin{aligned}
&-K\delta p(r_2)(1+\delta) + K\delta \\
&-K(\delta p(r_2) + \delta^2 p(r_2)) + K\delta \\
&K(\delta - \delta p(r_2) - \delta^2 p(r_2)) \\
&K(1 - p(r_2)(1+\delta))
\end{aligned}
$$

where if $p(r_2)(1+\delta) < 1$, the cost increases with increasing K. □

Lemma E.2: For all schedules in (ii), the cost *decreases* with *increasing* K.

Proof:

$$
\begin{aligned}
&-2Kp(r_2)(1+\delta) + K\delta + K\delta p(r_2)(1+\delta) \\
&-2Kp(r_2) - 2K\delta p(r_2) + K\delta + K\delta p(r_2) + K\delta^2 p(r_2) \\
&-2Kp(r_2) - K\delta p(r_2) + K\delta + K\delta^2 p(r_2) \\
&K(\delta^2 p(r_2) - \delta p(r_2) - 2p(r_2) + \delta) \\
&K(p(r_2)(\delta^2 - \delta - 2) + \delta)
\end{aligned}
$$

where $p(r_2)(\delta^2 - \delta - 2) + \delta < 0$ since $\delta \leq p(r_2)$ and the polynomial $\delta^2 - \delta - 2$ is less than or equal to -2 for $0 \leq \delta \leq 1$. □

Lemma E.3: For all schedules in (iii), the cost *decreases* with *increasing* K.

Proof:

$$
\begin{aligned}
&-K\delta(1+p(r_2)) - K\delta(1+\delta) + K\delta^2(1+p(r_2)) + K\delta \\
&-K\delta - K\delta p(r_2) - K\delta - K\delta^2 + K\delta^2 + K\delta^2 p(r_2) + K\delta \\
&-K\delta p(r_2) - K\delta + K\delta^2 p(r_2) \\
&K(\delta^2 p(r_2) - \delta p(r_2) - \delta)
\end{aligned}
$$

where $\delta^2 p(r_2) - \delta p(r_2) - \delta < 0$, since $\delta^2 - \delta$ is less than zero for $0 \leq \delta \leq 1$. □

Consider the schedules that are one step away from S_0: the schedule S_1 that pushes only from r_2, the schedule S_2 that pushes only from r_1, and the schedule S_3 that push from r_1, r_2, and any one of the remaining ranges. According to the above formulation, from (i) we get S_0 for $K = 0$ and S_3 for $K = 1$, from (ii) we get S_1 for $K = 0$, and from (iii) we get S_2 for $K = 0$. Note that from (iv) we cannot get a schedule that is one step away from S_0. Among the schedules in sets (i), (ii), (iii), and (iv), the least costly schedule is S_0 in (i), S_4 for $K = T$ in (ii), S_5 for $K = T$ in (iii), and the singleton S_6 in (iv).

We illustrate all these concepts with an example. For $T = 2$, we have four ranges as r_1, r_2, r_3, and r_4, and two queries as $Q_3 : \{r_1, r_2, r_3\}$ and $Q_4 : \{r_1, r_2, r_4\}$. We denote the schedule 1100 as S_0. The neighbors of S_0 are $S_1 = 0100$, $S_2 = 1000$, and the set $\{1101, 1110\}$. The schedules $\mathcal{S}$ we consider altogether are: $S_0 = 1100$, $S_1 = 0100$, $S_2 = 1000$, $S_3 = 1101$ or $S_3 = 1110$, $S_4 = 0111$, $S_5 = 1011$, and $S_6 = 0011$. We first show that S_0 is the least costly schedule for $p(r_2) = \delta$. The costs associated with each schedule $\in \mathcal{S}$ are:

$$
\begin{aligned}
cost(S_0) &= T\delta p(r_2)(1+\delta) + p(r_2) + \delta \\
cost(S_1) &= 2Tp(r_2)(1+\delta) + p(r_2) \\
cost(S_2) &= T\delta(1+p(r_2)) + T\delta(1+\delta) + \delta \\
cost(S_3) &= (T-1)\delta p(r_2)(1+\delta) + \delta + p(r_2) + \delta \\
cost(S_4) &= p(r_2) + T\delta + T\delta p(r_2)(1+\delta) \\
cost(S_5) &= T\delta^2(1+p(r_2)) + T\delta + \delta \\
cost(S_6) &= T\delta(2+\delta+p(r_2)) + T\delta
\end{aligned}
$$

The schedule S_3 is in the same group (i) with S_0. Therefore, it has a larger cost because of a larger K. Consider the schedule S_4. Certainly, $cost(S_0) \leq cost(S_4)$ since $\delta < T\delta$ for $1 < T$. Note that this inequality is independent of the value of $p(r_2)$. Since S_4 is the least costly in (ii), all other schedules including S_1 in (ii) have a larger cost than S_0. Among schedules in (iii), S_5 is the least costly schedule.

$cost(S_0)$	$cost(S_5)$
$T\delta p(r_2)(1+\delta) + p(r_2) + \delta$	$T\delta^2(1+p(r_2)) + T\delta + \delta$
$T\delta^2(1+\delta) + \delta$	$T\delta^2(1+\delta) + T\delta$
δ	$T\delta$

where RHS is always larger for $1 < T$. This implies that S_2 in (iii) also has a larger cost than S_0. The schedule S_5 has a smaller cost than the schedule S_6, which is the only member in (iv) as shown below:

$cost(S_5)$	$cost(S_6)$
$T\delta^2(1+p(r_2)) + T\delta + \delta$	$T\delta(2+\delta+p(r_2)) + T\delta$
$T\delta(1+p(r_2)) + T + 1$	$T(2+\delta+p(r_2)) + T$
$T\delta + T\delta p(r_2) + T + 1$	$2T + T\delta + Tp(r_2) + T$
$T\delta p(r_2) + 1$	$2T + Tp(r_2)$

since $\delta \leq 1$ and $1 \leq T$, RHS is larger than LHS. This implies that S_0 has a smaller cost than the schedule S_6. Also note that the inequality between S_5 and S_6 does not depend on $p(r_2)$.

Up to now, we have shown that S_0 can always stay larger than its immediate neighbors for a specific system configuration. Next we will show that the schedule S_5 will have a smaller cost than S_0 for some value of $p(r_2)$. In order to find this cut-off point for $p(r_2)$, we compare the schedules S_0 and S_5 as follows:

$cost(S_0)$	$cost(S_5)$
$T\delta p(r_2)(1+\delta) + p(r_2) + \delta$	$T\delta^2(1+p(r_2)) + T\delta + \delta$
$\delta T p(r_2) + p(r_2)$	$\delta^2 T + \delta T$
$p(r_2)(1+\delta T)$	$\delta^2 T + \delta T$
$p(r_2) = \frac{\delta^2 T + \delta T}{1+\delta T}$	

For example, let δ be 0.1. For $T = 2$, we have the cut-off point at $0.22/1.2$, which is 0.1833. So, when we keep increasing $p(r_2)$, starting from $p(r_2) = \delta = 0.1$, we will have the least costly schedule S_5 when $p(r_2) = 0.1833$, but all schedules in the neighborhood of S_0 will have larger costs than S_5. □