432

A Dual Framework and Algorithms for Targeted Data Delivery

Haggai Roitman, Avigdor Gal
Technion - IIT
Haifa 32000 Israel
{avigal@ie, haggair@tx}.technion.ac.il

Louiqa Raschid
University of Maryland
College Park MD 20742
louiqa@umiacs.umd.edu

Laura Bright
Portland State University
Portland OR 97207
bright@cs.pdx.edu

Abstract

A variety of emerging wide area applications challenge existing techniques for data delivery to users and applications accessing data from multiple autonomous servers. In this paper, we develop a framework for comparing pull based solutions and present dual optimization approaches. The first approach maximizes user utility while satisfying constraints on the usage of system resources. The second approach satisfies the utility of user profiles while minimizing the usage of system resources. We present a static optimal solution (SUP) for the latter approach and formally identify sufficient conditions for SUP to be optimal for both. A shortcoming of static solutions to pull-based delivery is that they cannot adapt to the dynamic behavior of wide area applications. Therefore, we present an adaptive algorithm (fbSUP) and show how it can incorporate feedback to improve user utility with only a moderate increase in resource utilization. Using real and synthetic data traces, we analyze the behavior of SUP and fbSUP under various update models.

1 Introduction

The diversity of data sources and Web services currently available on the Internet and the computational Grid, as well as the diversity of clients and application requirements poses significant infrastructure challenges. In this paper, we address the task of targeted data delivery. Users may have specific requirements for data delivery, e.g., how frequently or under what conditions they wish to be alerted about update events or update values, or their tolerance to delays or stale information. Initially these users were humans but they are being replaced by decision agents. The challenge is to deliver relevant data to a client at the desired time, while conserving system resources. We consider a number of scenarios including RSS news feeds, stock prices and auctions on the commercial Internet, scientific datasets and Grid computational resources. We consider an architecture of a proxy server that is managing a set of user profiles that are specified with respect to a set of remote autonomous servers. Push, pull and hybrid protocols have been used in data delivery, and we discuss data delivery challenges for these scenarios. We then focus on pull based resource monitoring and satisfying user profiles and define two optimization problems and present solutions.

Consider the commercial Internet where agents may be monitoring multiple sources. While push based protocols may be exploited to satisfy user profiles, targeted data delivery may require additional support. For example, there may be a mismatch between a simple profile that a server supports via push and the more complex profile of a decision making agent. A complex profile involving multiple servers may also require pull based resource monitoring, e.g., a profile that check a change in a stock price soon after a financial report is released cannot be supported by push from a single server. A decision agent may also not wish to reveal her profile for privacy or other considerations.

The scenario of RSS feeds is growing in popularity and it is supported by a pull based protocol. As the number of users and servers grow, targeted data delivery by a proxy can better manage system resources. In addition, the use of profiles could lower the load on RSS servers by accessing them only when a user profile is satisfied. A related application is the personalization of individual Web pages; these are typically supported by continuous queries which have similar requirements to satisfying user profiles.

Many scientific datasets are being made available and the by-products of scientific computations are increasingly being cached and re-used. In addition, the number of sites with computational resources for on demand computing is increasing. Grid resource monitors such as NWS [12] have been successful in monitoring and predicting the behavior of individual resources, but such systems have not been designed for scalability. Push based solutions are unlikely to scale to these applications. Users will have specific requirements for datasets, as well as specific configurations of resources. The large number of resources and the specialized user profiles will challenge a proxy server to actively and efficiently monitor resources and satisfy user profiles. To summarize, there are many scenarios where targeted data delivery requires a proxy server to efficiently monitor multiple servers to satisfy user profiles.

Much of the existing research in pull-based data delivery (e.g., [4, 9]) casts the problem of data delivery as follows: *Given some set of limited system resources, solve a (static) optimization problem to maximize the utility of a set of user profiles*. We refer to this problem as $OptMon_1$.

Based on our motivating scenarios, we propose a framework where we consider the dual of the previous optimization problem, as follows: *Given some set of user profiles, solve a (static) optimization problem to minimize the consumption of system resources while satisfying all user profiles.* We label this problem as $OptMon_2$; it will be formally defined in Section 2.2. We present an optimal (static) algorithm for this problem, namely *Satisfy User Profiles (SUP)*. We note that while our solution SUP is a pull based solution, it can be modified to work with hybrid push-pull protocols.

A limitation of the $OptMon_1$ formulation is that algorithms in this class typically assume a priori estimates of resource constraints. They do not attempt to determine an adequate level of resource consumption appropriate to satisfy a set of profiles given the update patterns of servers. Given the diversity of Web and Grid resources, and the complexity of user profiles, estimating the needed system resources is critical. We present some complex profiles in this paper which will illustrate the difficulty in determining a priori the appropriate level of resources that are needed. It is generally not known a priori how many times we need to *probe sources*. An estimate that is too low will fail to satisfy the user profile, while an estimate that is too high may result in excessive and wasteful probes. Conversely, solutions to $OptMon_1$ also have not attempted to reduce resource consumption, even if doing so would not negatively impact client utility. We will specify in Section 3 the conditions under which solutions to $OptMon_2$ are guaranteed to satisfy all user profiles (maximize utility), while minimizing total number of probes.

A more serious limitation is that most prior work provides *static* solutions to the problem of maximizing utility subject to system constraints, and cannot easily adapt to changes in source behavior. Existing solutions typically rely heavily on the existence of an accurate update model of sources. Unfortunately such models may not be completely accurate. Further, source behavior may change over time. The autonomy and diversity of Web and Grid resources invariably means that any choice of values for these parameters may not be appropriately chosen. Further, these parameters will need to be updated continuously. Also, the model of updates at the source may not be perfect. Finally, we can expect an update trace to have stochastic behavior, correlations, and bursts. Any offline solution must be able to *adapt* to reflect the online changing behavior of sources. Adaptations may include exploiting feedback from probes; changing the model of updates; and even changing the policy that is used to determine the next probe, e.g., adapting between algorithms for $OptMon_1$ and $OptMon_2$. In this work we present a solution *fbSUP* that incorporates feedback and dynamically changes the scheduling for probing. fbSUP assumes that the underlying model is accurate and utilizes feedback to adapt to stochastic variations.

Our contributions can be summarized as follows:

- At a conceptual level, we present a framework of dual offline optimization problems $OptMon_1$ and $OptMon_2$.
- At an algorithmic level, we present a (static) optimal solution for $OptMon_2$, namely *Satisfy User Profiles (SUP)*.
- We show that for a strict utility function (Section 2) and with sufficient resources, SUP is guaranteed to find an optimal solution to $OptMon_2$ and it does so by consuming a minimal number of probes. Thus, in this situation SUP is optimal for $OptMon_1$ as well.
- Using real trace data from an RSS server and synthetic data, and example profiles, we evaluate the performance of solutions to $OptMon_1$ and $OptMon_2$.
- We present a generic adaptive solution to SUP, namely fbSUP. The specific adaptation that we consider is exploiting feedback from probing the server to change the decision of when to probe next. We demonstrate empirically that fbSUP improves on the utility of SUP with a moderate overhead of additional probes.

The rest of the paper is organized as follows. Section 2 provides a framework for targeted data delivery. We next introduce SUP, an optimal static algorithm for solving an $OptMon_2$ problem (Section 3). We present our empirical analysis in Section 4 and then offer an online algorithm, fbSUP, to improve the performance of SUP (Section 5). We conclude with a description of related work (Section 6) and conclusion (Section 7).

2 Framework for Targeted Data Delivery

We now present our framework for targeted data delivery. We define a specification language for user profiles; it can be used in conjunction with push or pull based delivery. We then focus on pull-based methods and introduce dual offline optimization problems, $OptMon_1$ and $OptMon_2$. We then discuss schedules and the utility of probing.

Let $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ be a set of pages, taken from various page classes (e.g., eBay, Yahoo! RSS pages, etc.) We denote by $\mathcal{D}_i$ the class of page P_i. Let $\mathcal{T}$ be an epoch and let $\{T_1, T_2, ..., T_N\}$ be a set of chronons (time points) in $\mathcal{T}$. A schedule $S = \{s_{i,j}\}_{i=1...n, j=1...N}$ is a set of binary decision variables, set to 1 if page P_i is probed at time T_j and 0 otherwise. Let $\mathcal{S}$ be the set of all possible schedules.

2.1 User Profiles and the Monitoring Task

User profiles are declarative specifications of goals for data delivery. Our profile language consists of three parts, namely Domain, Notification, and Profile. Domain is a set of classes in which users have some interest. The domain includes two class types, namely "Query classes," classes that users wish to monitor and "Trigger classes" that are used to determine when some monitoring action should be executed. For example, the condition for monitoring a page from class D_1 (say `stock prices`) may be based on updates to a page from class D_2 (say, `financial news reports`). Query classes and trigger classes may overlap. We present here a simple domain creation example in our profile language, named "*RSS_Feeds*" with three fields, title, description, and publication date. A full-scale profile language discussion is beyond the scope of this paper.

```
CREATE DOMAIN "RSS_Feeds" AS
CLASS::RSS(channel.item.title:String,
channel.item.description:String,
channel.item.pubDate:String);
```

A profile is a triplet. It contains a list of predefined domains such as a traffic domain or a weather domain, a set of notification rules that are defined over the domains (see below), and a set of users that are associated with the profile. A notification rule may be associated with different profiles. The following profile is defined for user *u025487* over the domain of *RSS feeds*, using SQL as its query language:

```
CREATE PROFILE "RSS_Monitoring" AS
(DOMAIN "RSS_Feeds",
LANGUAGE "SQL",
USER "u025487");
```

A notification rule η is a quadruple of the form $\langle Q, Tr, \mathcal{T}, U\rangle$. Q is a query written in any standard query language, specifying the Query classes (in some domain) that are to be monitored. *Tr* is a triggering expression and can include a triggering event and a condition that must be satisfied for monitoring to be initiated; both are defined for the Trigger class(es). $\mathcal{T}$ is the period of time during which the notification rule needs to be supported. U is a utility function specifying the utility gained from notifications of Q. For each event that occurs, the notification has two possible states, namely "Triggered" and "Executable." A notification enters the Triggered state when an event specified in *Tr* occurs; the condition of *Tr* is then evaluated. If this condition is `true`, then the notification goes to the Executable state (for that event). A notification remains executable as long as the condition is true. The period in which a notification rule is executable was referred to in the literature as *life* [9]. Two examples of life we shall use in this paper are `overwrite`, in which an upade is available for monitoring only until the next update to the same page occurs. A more relaxed life setting is called *window(Y)*, for which an update can be monitored up to Y chronons after it has occured.

The period of time on which the notification is executable for some event defines a possible "execution interval," duing which monitoring should take place. That means that the query part of a notification that defines the monitoring task should be executed. Each notification rule η is associated with a set of execution intervals $EI(\eta)$. For each $I \in EI(\eta)$ we define $\tau(I)$ as the times T_j on which the notification is executable for interval I. It is worth nothing that execution intervals of a notification rule may overlap, thus the execution of notification query may occur at the same time for two or more events that cause the notification to become executable.

As an example, suppose the user would like to be notified every time there are X new RSS feeds. The notification rule `Num_Update_Watch`, to be associated with *RSS_Monitoring* profile is defined as follows:

```
INSERT NOTIFICATION "Num_Update_Watch"
INTO "RSS_Monitoring"
SET QUERY "SELECT channel.item.title,channel.item.description
        FROM RSS"
SET TRIGGER "ON INSERT TO RSS
      WHEN COUNT(*) % X =0"
START NOW
END "30 days" + NOW
SET UTILITY STRICT
```

Monitoring can be done using one of three methods, namely push-based, pull-based, or hybrid. With push-based monitoring the server pushes updates to clients, providing guarantees with respect to data freshness at a possibly considerable overhead at the server. With pull-based monitoring, content is delivered upon request, reducing overhead at servers, with limited effectiveness in estimating object freshness. The hybrid approach combined push and pull, either based on resource constraints [6] or role definition.

For the latter, consider the user profile language we have presented. Here, it is possible that servers of trigger classes will push data to clients, while data regarding query classes will be monitored by pulling content from servers once a notification rule is satisfied. As another example for the hybrid approach, consider a three-layer architecture, in which a mediator is positioned between clients and servers. The mediator can monitor servers by periodically pulling their content, and determine when to push data to clients based on their content delivery profiles.

In the rest of the paper we focus on challenges in pull-based monitoring. We start with detailing the impact on pull-based monitoring on clients and introduce the dual optimization problem.

2.2 Framework of Dual Offline Optimization Problems

There are two, principally different, approaches to support pull-based targeted data delivery. One formulation $OptMon_1$ assumes an a priori independent assignment of system resources to this task, e.g., an upperbound on bandwidth for probing. The task is to maximize user benefit under such constraints. It is as follows:

$$\begin{aligned}&\text{maximize user utility}\\ &s.t.\ \text{satisfying system constraints}\end{aligned} \tag{1}$$

For example, in [9] $OptMon_1$ involves a system resource constraint of M, the maximum number of probes per chronon for all pages in $\mathcal{P}$. One can specify system resources in terms of number of probes, assuming each probe has an overhead of opening a TCP/IP channel of communication, downloading information from the server, deciding on the timing of the next probe, etc. User utility can be parameterized to represent the amount of tolerance a user has to delayed delivery.

We propose a dual formulation $OptMon_2$, which reverses the roles of user utility and system constraints. It assumes that the system resources that will be consumed to satisfy *user profiles* should be determined by the specific profiles and the environment, e.g., the model of updates. Thus, we do not assume an a priori limitation of system resources. $OptMon_2$ is the following optimization problem:

$$\begin{aligned}&\text{minimize system resource usage}\\ &s.t.\ \text{satisfying user profiles}\end{aligned} \tag{2}$$

The dual problems are inherently different. To illustrate this, consider two resource constraints, namely M (the maximum number of probes per chronon) and N, the number of chronons in $\mathcal{T}$. The total number of available probes in $\mathcal{T}$ is $N \cdot M$. The behavior of all solutions to $OptMon_1$ will be controlled by the values for M and N. For example, a choice of a smaller chronon size results in larger N and an increased utilization of probes (system resources) by any solution to $OptMon_1$. On the other hand, solutions to $OptMon_2$ can benefit from parameter settings, but the parameter values do not control their behavior. For example, a smaller chronon size and larger N allows a solution to $OptMon_2$ to probe resources at a finer level of chronon granularity; however, the minimization of resource utilization would ensure that resource consumption will not increase in vain. To summarize, solutions to the two problems cannot be compared directly. No solution for one problem can dominate a solution to the other, for all possible problem instances.

2.2.1 Roadmap for Investigating the Dual Problem

In Section 3 we introduce SUP, an efficient algorithm for the $OptMon_2$ challenge of targeted data delivery, given a user profile. SUP is guaranteed to minimize system resources while satisfying a user profile. SUP is an offline algorithm. It determines a probing schedule given an a priori update model of resources. Therefore, in run-time it may not be optimal due to two main problems. First, an update model for pull-based monitoring is necessarily stochastic and therefore is subject to variations, due to the model variance. Second, it is possible that the update model is inaccurate to start with, which means that replacing it with another, more accurate update model would yield better schedule. In this research, we address the first problem by offering *fbSUP*, an online algorithm that makes use of feedback to tune its schedule. We defer the algorithmic solution of the second problem to an extended version of this work.

2.3 Schedules and the Utility of Probing

Let $\mathcal{N}_k$ be the set of notification rules of profile p_k. Let $\eta \in \mathcal{N}_k$ be a notification rule that utilizes classes from p_k domain and let Q^η be the set of all pages in $\mathcal{P}$ that are in the domain of p_k. We now define the satisfiability of a schedule with respect to η as follows:

Definition 2.1. *Let $S \in \mathcal{S}$ be a schedule, η be a notification rule with Q^η, and $\mathcal{T}$ be an epoch with N chronons. S is said to* satisfy *η in $\mathcal{T}$ (denoted $S \models_{\mathcal{T}} \eta$) if $\forall I \in EI(\eta) \forall P_i \in Q^\eta (\exists T_j \in \tau(I) : s_{i,j} = 1)$.*

Definition 2.1 requires that in each execution interval, every page in Q^η is probed at least once. Whenever it becomes clear from the context, we use $S \models \eta$ instead of $S \models_{\mathcal{T}} \eta$. This definition is easily extended to a profile and a set of profiles, as follows:

Definition 2.2. *Let $S \in \mathcal{S}$ be a schedule, $\{p_1, p_2, ..., p_m\}$ be a set of profiles, and $\mathcal{T}$ be an epoch with N chronons.*

S is said to satisfy $p_k \in \{p_1, p_2, ..., p_m\}$ *(denoted $S \models p_k$) if for each notification rule $\eta \in \mathcal{N}_k$, $S \models \eta$.*

S is said to satisfy $\{p_1, p_2, ..., p_m\}$ *(denoted $S \models \{p_1, p_2, ..., p_m\}$) if for each profile $p_k \in \{p_1, p_2, ..., p_m\}$, $S \models p_k$.*

Given a notification rule $\eta \in \mathcal{N}_k$ and a page $P_i \in \cup_{\eta \in \mathcal{N}_k} Q^\eta$, a utility function $u(P_i, \eta, T_j)$ describes the utility of probing a page P_i at chronon T_j. Intuitively, probing a page P at time T is useful (and therefore should receive a positive utility value) if it belongs to a class that is referred to in the Query part of the notification rule and if the condition in the Trigger part of that profile holds. It is important to emphasize again the difference of roles between the Query part and the Trigger part of the profile. In particular, probing a page P is useful only if the data required by a profile (specified in the Query part) can be found at P.

$u(P_i, \eta, T_j)$ is derived by assigning positive utility when a condition is satisfied, and a utility of 0 otherwise. u is defined to be *strict* if it satisfies the following condition:

$$u(P_i, \eta, T_j) = \begin{cases} w & T_j \in \cup_{I \in EI(\eta)} \tau(I) \wedge P_i \in Q^\eta \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

From now on we shall assume binary utility, i.e., $w = 1$. Example of strict utility functions include [9], *uniform* (where utility is independent of delay) and *sliding window* (where utility is 1 within the window and 0 out of it). Examples of non strict utility functions are non-linear decay functions. For simplicity, we shall restrict ourselves to strict utility functions.

3 An Optimal Static Algorithm SUP

Let $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ be a set of n pages, $\{T_1, T_2, ..., T_N\}$ be a set of chronons in an epoch $\mathcal{T}$, and $S = \{s_{i,j}\} \in \mathcal{S}$ be a schedule. Let $\{p_1, p_2, ..., p_m\}$ be a set of user profiles. $OptMon_2$ is the following optimization problem:

$$\text{minimize} \sum_{P_i \in \mathcal{P}, T_j \in \mathcal{T}} s_{i,j}$$

$$s.t.\ S \models \{p_1, p_2, ..., p_m\} \tag{4}$$

The expected utility U accrued by executing monitoring schedule S in an epoch $\mathcal{T}$, is given by:

$$U(S) = \sum_{\eta \in \cup_{l=1}^m \mathcal{N}_l} \sum_{I \in I(\eta)} \sum_{P_i \in Q^\eta}^{m} \min\left(1, \sum_{T_j \in I} s_{i,j} u(P_i, T_j, \eta)\right) \tag{5}$$

The innermost summation ensures that utility is accumulated whenever a probe is performed within an execution interval. The utility cannot be more than 1 since probing a page more than once within the same execution interval does not increase its utility. The utility is summed over all execution intervals, all relevant pages, and over all notification rules in a profile.

3.1 The SUP Algorithm

Recall that a notification rule η is associated with a set of pages Q^η. Given a notification rule η and the set of its execution intervals $EI(\eta)$, SUP identifies the set of pages $Q_I^\eta \subseteq Q^\eta$ that must be probed in an execution interval I. We present a static algorithm SUP for solving $OptMon_1$. By static we mean that the schedule is determined a-priori. Later, in Section 5, we show an adaptive algorithm that can exploit feedback.

Algorithm 1 (SUP)

```
Input: $\mathcal{P}, \mathcal{T}, \mathcal{N} = \cup_{l=1}^m \mathcal{N}_l$
Output: $S = \{s_{i,j}\}$
(1)  For all pages $P_i \in \mathcal{P}$ and chronons $T_j \in \mathcal{T}$:
(2)      Initialize $s_{i,j} \leftarrow 0$.
(3)  For $l = 1$ to $|\mathcal{N}|$:
(4)      $T_l = \min_{I \in EI(\eta)}\{\max \tau(I)\}$
         /* $T_l$, is the last chronon of the first */
         /* execution interval of notification rule $\eta_l$*/
(5)  repeat
(6)      $\tau_j = \min_{l=1}^{|\mathcal{N}|}(T^l)$
         /* $\tau_j$ is the earliest chronon for which a notification rule */
         /* is executable and when SUP will probe*/
(7)      $\eta = argmin_{l=1}^{|\mathcal{N}|}(T^k)$
         /* $\eta$ is the notification rule whose */
         /* pages in $Q_I^\eta$ need to be probed*/
(8)      $I = argmin_{I \in EI(\eta)}\{\max \tau(I)\}$
(9)      For all $P_i \in Q_I^\eta$:
(10)         set $s_{i,j} \leftarrow 1$
(11)     For $k = 1$ to $|\mathcal{N}|$:
(12)         *UpdateNotificationEIs*$(\tau_j, \mathcal{N}_k)$
(13)     $T_l = \min_{I \in EI(\eta)}\{\max \tau(I)\}$
(14) until $T^l > N$
```

The algorithm builds a schedule iteratively. It starts with an empty schedule ($\forall s_{i,j} \in S, s_{i,j} = 0$) and repeatedly adds probes. The "for loop" in lines 3-4 generates an initial probing schedule, where the last chronon in the first $I \in EI(\eta)$ is picked to execute the probe. Lines 5-8 determine the earliest chronon in which a probe is to be made, the notification rule associated with this probe, and the specific execution interval. All pages that belong to classes in the query part of that notification rule are probed (lines 9-10).

In line 12, the algorithm uses a routine, *UpdateNotificationEIs,* to ensure that pages that belong to overlapping intervals are only probed once. Let $l = \eta$ be the assignment in line 7 of the algorithm. η is the notification rule whose execution interval I is processed at time j, and all pages that belong to classes in Q_I^η at time j are scheduled for probing. Given an execution interval I' of a notification rule η', this routine removes from $Q_{I'}^{\eta'}$ the (possibly empty) class set $Q_I^\eta \cap Q_{I'}^{\eta'}$ if $\tau(I) \cap \tau(I') \neq \emptyset$. By doing so, we ensure that pages that belong to overlapping execution intervals will be probed only once. In addition, this routine removes any execution interval I for which $Q_I^\eta = \emptyset$, allowing lines 4 and 13 to consider only execution intervals for which monitoring is still needed.

The process continues until the end of the epoch. Generally speaking, a new probe is set for a page at the last possible chronon where a notification remains executable. That is, it is deferred to the last possible chronon where the utility is still 1. This, combined with the use of the routine *UpdateNotificationEIs*, is needed to develop an optimal schedule, in terms of resource utilization.

The following theorem ensures the optimal outcome of the algorithm.

Theorem 3.1. *Let $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ be a set of n pages, $\{T_1, T_2, ..., T_N\}$ be a set of chronons in an epoch $\mathcal{T}$, and $S = \{s_{i,j}\}$ be a monitoring schedule, generated by Algorithm SUP, with $\sum_{P_i \in \mathcal{P}, T_j \in \mathcal{T}} s_{i,j} = K$. Let $S' \in \mathcal{S}$, $S' \models \{p_1, p_2, ..., p_n\}$ with $\sum_{P_i \in \mathcal{P}, T_j \in \mathcal{T}} s'_{i,j} = K'$. Then $K \leq K'$.*

Proof (Sketch) We provide a sketch of a constructive proof, in which S' is modified into S without increasing the number of probes. We look at the first chronon j for which $s_{i,j} \neq s'_{i,j}$, for some page P_i. In case $s_{i,j} = 0$ and $s'_{i,j} = 1$, we identify $j^* > j$ such that j and j^* are within the same execution interval of a notification rule η such that $P_i \in Q^\eta$ and $s_{i,j^*} = 1$. If there is no such j^*, it means that P_i was probed already by both S and S' (to satisfy some profile) so we set $s'_{i,j} = 0$. If we find such j^* we set $s'_{i,j} = 0$ and $s'_{i,j^*} = 1$. In the first case we decrease K' by 1, while in the second case we either do not change K' (in case s'_{i,j^*} was originally set to 0) or we decrease K' by 1 (in case s'_{i,j^*} was originally set to 1). We continue iteratively until S' becomes S. Due to the construction, we get that $K \leq K'$. ■

Probing at the last possible chronon ensures an optimal usage of resources while still satisfying user profiles. However, due to the stochastic nature of the process, probing later may decrease the probability of satisfying the profile. This is true for example with hard deadlines; once the deadline is passed, the utility is 0. Determining an optimal probing point, i.e., the one that maximizes the probability of satisfying the profile depends on the stochastic process of choice, and is itself an interesting optimization problem. We defer this analysis to an extended version of this work. In Section 5 we introduce an adaptive algorithm that improves the online performance of SUP.

SUP accesses $O(K)$ execution intervals, where K is the number of total probes in a schedule, bounded by Nn. We expect, however, K to be much smaller than Nn, since K serves as a measure of the amount of data clients expect to receive during the monitoring process.

Theorem 3.1 shows that Algorithm SUP indeed minimizes $\sum_{P_i \in \mathcal{P}, T_j \in \mathcal{T}} s_{i,j}$. The following theorem (which proof is immediate from Eq. 5) shows that the schedule generated by Algorithm 1 also has maximum utility for the class of strict utility functions (and hence can maximize utility while minimizing system resource consumption).

Theorem 3.2. *Let $\mathcal{P} = \{P_1, P_2, ..., P_n\}$ be a set of n pages, $\{T_1, T_2, ..., T_N\}$ be a set of chronons in an epoch $\mathcal{T}$, $\{p_1, ..., p_m\}$ be a set of user profiles and $S = \{s_{i,j}\}$ be a monitoring schedule, generated by Algorithm SUP, with an expected utility $U(S)$.*
If for any notification rule η in $\{p_1, ..., p_m\}$, $u(P_i, T_j, \eta)$ is strict, then $U(S) \geq U(S')$, for any schedule $S' = \{s'_{i,j}\} \neq S$.

Proof To maximize Eq. 5, a schedule has to schedule a probe in each execution interval. SUP does that and therefore $U(S)$ has maximum value. ■

3.2 A Comparison of $OptMon_1$ and $OptMon_2$

Generally speaking, the dual optimization problems $OptMon_1$ and $OptMon_2$ cannot be compared directly. Satisfying user profiles may violate system constraints and satisfying system constraints may fail to satisfy user profiles. However, Theorem 3.2 provides an interesting observation. Whenever the resources consumed by SUP satisfy the system constraints of $OptMon_1$, then SUP is guaranteed to solve the dual $OptMon_1$ and maximize user utility, while at the same time minimizing resource utilization.

As an example, consider an algorithm (e.g., [9]) that sets an upper limit M on the number of probes in a chronon for all pages. Assume that in the schedule of SUP, the **maximum** number of probes in any chronon does satisfy M. Since SUP utilizes in each chronon only the amount of probes that is needed to satisfy the profile expressions, the total number of probes will never exceed $N \times M$. The rigidity of $OptMon_1$ algorithms forces them to utilize all available resources. This could be more than the resource utilization of SUP.

Whenever strict utility functions are used, Algorithm SUP can serve as a basis for solving the dual problem $OptMon_1$. A schedule S, generated by SUP with no bound on system resource usage, and a set of desired system resource constraints, can be used as a starting point in solving $OptMon_1$. S can be used to avoid over-probing in chronons when less updates are expected. Resources may be allocated to chronons that are more update intensive. In this situation, SUP may serve as a tentative initial solution to the $OptMon_1$ problem, allowing local tuning at the cost of reduced utility. We defer a formal discussion of SUP under system constraints to an extended version of this paper.

4 Experiments

We now present empirical results on the behavior of SUP. We start in Section 4.1 with a description of the data sets and the experiment setup. We then analyze the impact of profile selection, life parameter, and update model on SUP performance (sections 4.2-4.4). We then present an empricial comparison of representatives of the dual optimization problems in Section 4.5.

4.1 Datasets and Experiment Setup

Dimension	Description	parameters
Data set	one real data set	see Table 2
	2 synthetic data sets	see Table 2
Notification rule	Num_Update_Watch	$X = 1, 2, 3, 4, 5$
life	overwrite	
	window	$Y =$ 0-100 chronons
update model	FPN	$Z = 1.0, 0.8, 0.6, 0.4$
	Poisson	λ

Table 1. Summary of the experiment parameters

We implemented SUP in Java, JDK version 1.4 and experimented with it on various data sets, profiles, life parameters, and update models. We consider a variety of traces of update events. These traces could be real traces, e.g., RSS feeds, or they could be synthetic traces. We consider two different update models, FPN and Poisson (to be discussed shortly) to model the arrival of new update events to these traces. For comparison purposes, we also implemented WIC to determine a schedule for $OptMon_1$ as described in [9]. Details are omitted for space considerations. Table 1 presents the various dimensions of our experiments. We next discuss each parameter in more details.

Dataset	Number of objects	Update events
RSS Feeds	103	1972
Synthetic Data 1	244	3754
Synthetic Data 2	792	4194

Table 2. Summary of the data sets

We used data from a real trace of `RSS Feeds`. We collected RSS news feeds from several Web sites such as CNN and Yahoo!. We have recorded the events of insertion of new feeds into the RSS files. We also generated two types of synthetic data. The first set simulates an epoch with three different Exponential inter-arrival intensity, medium (first half a day), low (next 2 days), and high (last half a day). This data set can model the arrival of bids in an auction (without the final bid sniping). The second data set has a stochastic cyclic model of one week, separating working days from weekends, and working hours from night hours. Such a model is typical for many applications [7], including posting to newsgroups, reservation data, etc. Here, it can be representative of an RSS data with varying update intensity. Both synthetic data sets were generated assuming an exponential inter-arrival time. Table 2 summarizes the properties of the three data sets. The epoch size vary from one data set to another. To compare them on even grounds, we have partitioned each epoch into 1000 chronons.

For the experiments, we used the profile `"RSS_Monitoring"` as defined in Section 2.1, with values of $X = 1, \cdots, 5$ for the `"Num_Update_Watch"` notification rule. As for the *life* parameter, we have experimented with `overwrite`, as well as `window` with $Y \in \{0, ...100\}\, chronons$.

An update model defines the monitoring system view of the update patterns of the data, which may or may not coincide with the actual update trace. To varify the impact of the client-side update modeling on SUP performance, we use two different update models to represent updates at the servers, as follows:

- **Poisson Update Model:** Following [7], we devised an update model as piecewise homogeneous Poisson processes. A Poisson process with instantaneous arrival rate $\lambda : \Re \rightarrow [0, \infty)$ models the occurrence of *update events*. The number of update events occurring in any interval $(s, f]$ is assumed to be a Poisson random variable with expected value $\Lambda(s, f) = \int_s^f \lambda(t)\, dt$. Due to limitation of this modeling technique (see [7] for details), the Poisson update model was applied to the synthetic data traces only.

- **False positives and False negatives (FPN) Update Model:** Following [9], we devised the FPN model. Given a stream of updates, a probability $p_{i,j}$ is assigned the value 1 if a page P_i is updated at time T_j. Once probabilities are defined, we add noise to the probability model, as follows. Given an error factor $Z \in [0, 1]$, the value of $p_{i,j}$ is switched from 1 to 0 with probability Z. Then, for each modified $p_{i,j}$, a new time point $T_{j'}$ is randomly selected and the value of $p_{i,j'}$ is set to 1. Note that FPN can be applied to any data trace, regardless of its true stochastic pattern.

With 3 datasets, 2 update models (and parameter variations for FPN) and varying life settings, there are a large number of possible experiment configurations. due to space consideration, in this work we restrict ourseleves to presenting results with the more "interesting" configurations.

To measure the performance of SUP, we use *effective utility*, as follows. For each experiment, an optimal schedule S^* for algorithm SUP was developed. Given a schedule S, the *effective utility* of an algorithm is calculated as the ratio of the utility gained by S divided by the utility of S^*.

4.2 Impact of profile selection

In our first experiment we report on the impact a choice of a profile has on the ability of SUP to perform online. In Section 3.1 SUP is proven to be optimal offline, yet stochastic variations may affect its online behavior. We chose a complex profile where the user wish to be notified only after X number of updates were accumulated. Note that this does not mean that the profile is less accurate and tolerates missing update events.

Figure 1 provides the results of experimenting with the five different profiles. We have set the update model to be $FPN(0.6)$. The x axis represents an increasing window size and each of the curves represent different X value. We present the results for two data sets, RSS and synthetic data 1. Synthetic data 2 show similar behavior as RSS.

For both datasets, the effective utility is reasonably high. It ranges from 90.7% to 99.4% for RSS and 71% to 80.5% for synthetic data 1. As we vary the parameter Y that is a part of the profile and control the window for reporting the event, the effective utility increases. The difficulty of supporting a complex profile is seen for both datasets. The effective utility for X=1 is higher than for X=5, reflecting that when the profile is required to determine when a fixed number of updates have accumulated, the error of misestimation also has a cumulative impact.

4.3 Impact of life parameter selection

The life parameter represents user tolerance towards the exact monitoring time. The most restricted parameter in our experiments is `window(0)`, requiring the schedule to monitoring at exactly the time of update. As window size increases, the schedule can probe at increasing distances from the actual update. `Overwrite` is restricted in settings where many updates occur at close proximity.

Figure 2 provides the results of experimenting with four different life parameters, `overwrite` and `window(Y)` with $Y = 0, 10, 20$ chronons. We have set the update model to be $FPN(0.6)$ again. The x axis represents different profiles (note the difference from the previous experiment) and each of the curves represent a different life parameter. We present the results for two synthetic data sets. Once again, synthetic data 2 and RSS show similar behavior.

Somewhat surprising, the worst performance is that of the `overwrite` life parameter. Note that the performance of `overwrite` in synthetic data 1 is much better than its counterpart. This indicates that with synthetic data 1 SUP is allowed more manuevering room to monitor properly, probably due to the way updates are spread across the epoch. As for the `window(Y)` life parameter, effective utility increases with Y, since a wider window allow a better chance for capturing updates.

4.4 Impact of update model selection

We study next how various parameter settings for the FPN model and the use of the Poisson model impact effective utility. Recall that as the Z FPN parameter is less than 1.0, we are introducing more stochastic variation in the update model. We present SUP performance for the `"Num_Update_Watch"` notification rule, with $X = 1, \cdots, 5$ and the `overwrite` life parameter. We use the synthetic data sets to illustrate our results.

In Figure 3, SUP has 100% utility for $Z = 1.0$, since SUP offline estimation has an accurate understanding of the update stream. As we modify the parameter to $Z = 0.4$, more variance is added, and effective utility deteriorates. Scheduled updates may come too early or too late. Generally speaking, Poisson seems to be the model with the highest variance, resulting in low effective utility. For synthetic data set 2, for all update models, the effective utility decreases as the number of updates in the notification rule increases from $X = 1, \cdots, 5$. Note, however, the relative stability of the Poisson model. Synthetic data set 1 demonstrates a different pattern, in which the change of effective utility, as X increases is less predictable.

4.5 $OptMon_1$ and $OptMon_2$

Recall that while $OptMon_1$ set hard constraints on system resources, $OptMon_2$ aims at minimizing its utilization; thus they cannot be compared directly. Further, $OptMon_2$ secures the full satisfaction of user specification (at least offline) while $OptMon_1$ can only aim at maximizing it. Despite their differences, we can compare them indirectly, using resource utilization of the different solutions and the utility for given resource utilization.

We compare SUP as a representative of $OptMon_2$ and WIC as a representative of $OptMon_1$. Figure 4 provides the resource utilitization and corresponding utility of both algorithms. The experiment used the `Synthetic Data 2` dataset. We add a parameter denoted M, used by WIC, to represent a system constraint on the total number of probes allowed per chronon. Figure 4(a) provides the analysis re-

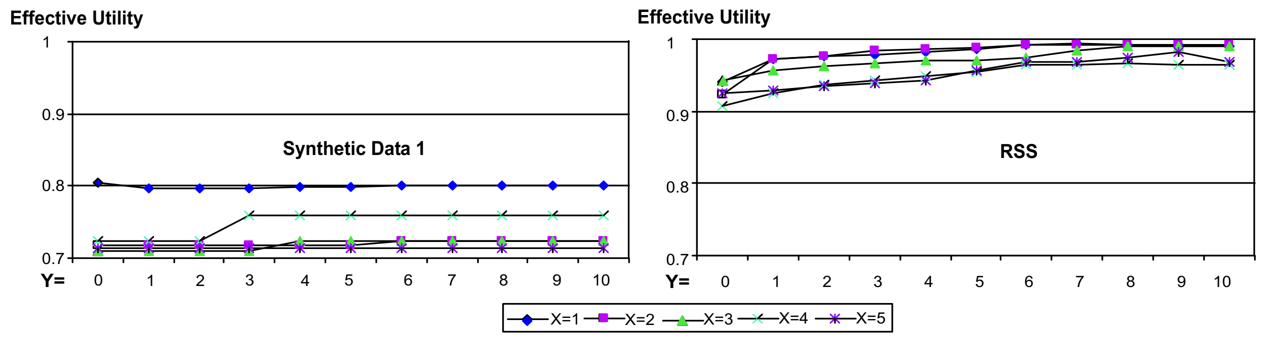

Figure 1. SUP performance for various profiles

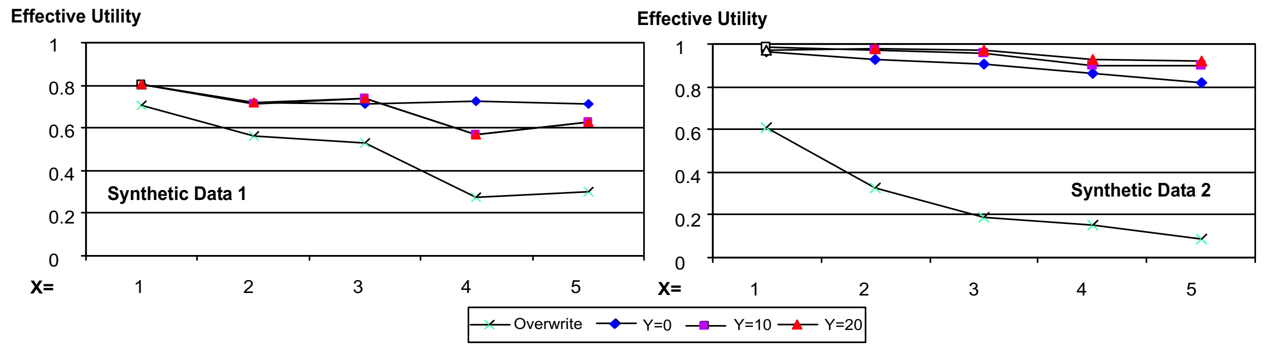

Figure 2. SUP performance for various life parameters

sults for $FPN(1.0)$, where updates occur at the expected update time as determined by the update model. Figure 4(b) provides the execution results, assuming a Poisson update model.

In Figure 4(a), SUP is represented by a single point in the graph; this is its optimal schedule with an effective utility of 1.0. The optimal number of probes for SUP is 1500 for this dataset. We study WIC under various parameter settings; we consider 500, 1000 and 3000 for the number of chronons in an epoch $\mathcal{T}$. The three curves WIC 500, WIC 1000 and WIC 3000 represent these parameter settings. We also varied the M level. The x axis represents the total number of probes ($N \cdot M$). Thus, for $N = 500$ chronons and $M = 30$, we have 15,000 probes. Similarly, with $N = 1000$ chronons and $M = 15$, we have 15,000 probes.

The effective utility of WIC approaches 1.0 with increased probing but does not reach it. We note that for 1500 probes, (where SUP has 1.0 effective utility in the ideal case), the effective utility of WIC is much below that of SUP.

We now consider the analysis of using the Poisson update model in Figure 4(b). The effective utility for SUP is about 0.41 (about 41% of the optimal). This too is for the same 1500 probes and is represented by a single point. WIC starts with low effective utility (less than 0.2) and as it increases the number of probes, the utility increases. In order to reach utility of 0.41 it requires more than 9000 probes, which is approximately 6 times higher than that of SUP. The relatively low effective utility indicates that predicting an update event using a static algorithm may not be very accurate. We will next present our adaptive algorithm, which allows changes to the a priori schedule using feedback.

5 Adaptive SUP

The optimal static algorithm SUP performs well assuming a good underlying update model. However, in practice, the SUP algorithm (or any algorithm that relies on a stochastic update model, for that matter) may perform poorly due to two main problems. First, the underlying update model that is assumed in the static calculation is stochastic in nature and therefore updates deviate from the expected update times. Second, it is possible that the underlying update model is incorrect, and the real data stream behaves differently than expected.

To tackle the first problem, we propose an algorithm fbSUP, a variant of SUP. fbSUP exploits feedback from

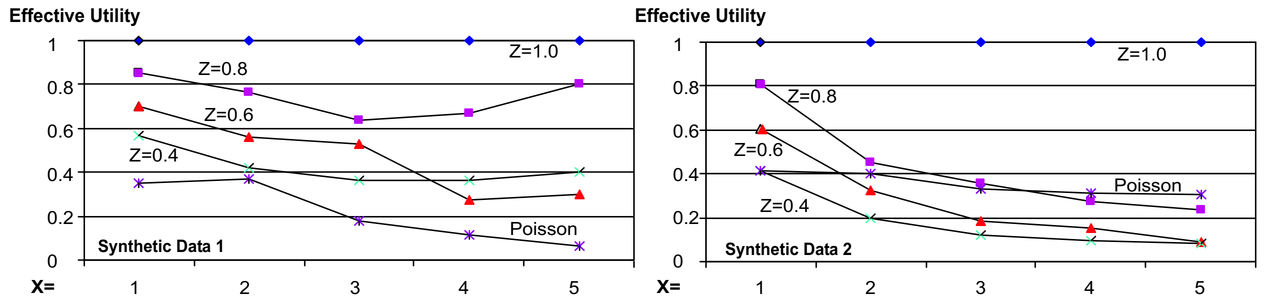

Figure 3. SUP performance for various update models

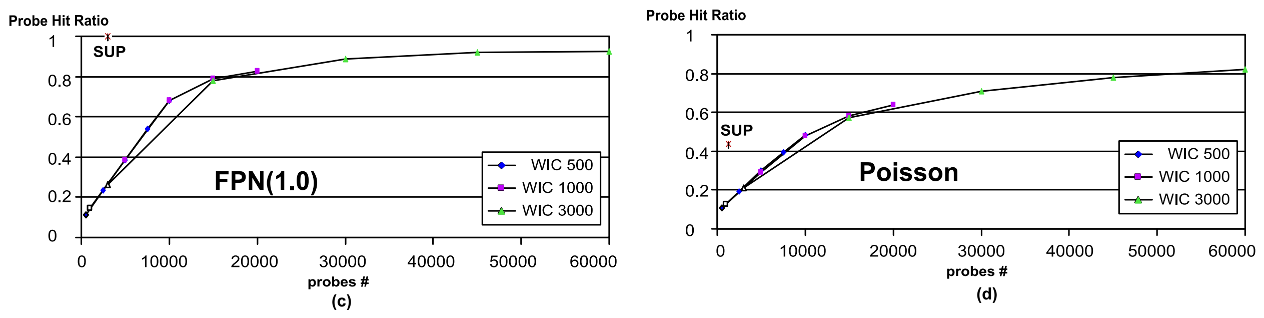

Figure 4. SUP and WIC for `Synthetic Data 2` dataset for (a) FPN(1) (b) Poisson

probes, and revises the probing schedule in a dynamic manner, after each probe. fbSUP responds to deviations in the expected update behavior of sources that it observes as a result of feedback from the probes, and it does so without requiring changes to any parameters.

fbSUP is independent of both the profile description and the specific details of the update model. The only requirement we make is that $\mathcal{T}$ be discrete.

5.1 fbSUP: SUP algorithm with feedback

We now present fbSUP, the adaptive SUP algorithm. Given a notification rule η and a chronon T, we define $C_\eta(T)$ to be a boolean variable, set to true if η is satisfied at time T. Intuitively the algorithm works as follows: Recall that given a client profile, SUP probes a source once in an execution interval. In the running example, that means that the expected number of updates has increased by X. fbSUP adds an extra probe whenever the recent probe has failed to satisfy $C_\eta(T)$, by generating a new execution interval in $EI\,(\eta)$. The algorithm for fbSUP is presented next.

Algorithm 2 (fbSUP)

```
Input: $P_i, T, \eta, S = \{s_{i,j}\}$
Output: $S' = \{s'_{i,j}\}$
(1) If $C_\eta(T) = False$ then:
(2)     $I = argmin_{I' \in EI(\eta)}\{\cup_{T' > T}\tau(I')\}$
(3)     $I^* =$ recompute execution interval for $\eta$ using feedback
(4)     If not exists $T_j \in \tau(I^*)$ such that $s_{i,j} = 1$ then:
(5)         $T_j = max(T \in \tau(I^*))$
(6)         set $s_{i,j} \leftarrow 1$
```

Line 2 identifies the nearest execution interval I to time T. Such execution interval is still current (that is $T \in \tau(I)$), based on the way SUP assigns probes to execution intervals. Clearly, no more probes, beyond the probe at time T is scheduled for I, due to the offline optimality of SUP. Therefore, fbSUP recomputes a new execution interval, based on the feedback it receives from the monitoring task (line 3). In lines 4-6 we verify that no other probe is scheduled for this page and then set a new monitoring task at the end of the new execution interval.

As an example, consider our case study notification rule and assume that at the time of monitoring, only $l < X$ updates occur. fbSUP generates a new execution interval, checking for $X - l$ updates ahead.

5.2 Discussion

fbSUP refrains from changing the update model as a method of adaptation. While changing the update model can increase the flexibility of the algorithm, we believe this approach has two main limitations. First, it does not seem to be theoretically sound. fbSUP uses feedback to learn of the natural variation of the stochastic model, rather than learning a new update model. Second, recomputing parameter values at every chronon increases the complexity of the algorithm, requires extensive bookkeeping, and could delay probes in some situations. Thus, this online approach to recompute parameter values does not optimize resource consumption and could potentially result in excessive probing.

5.3 Experiments with fbSUP

We performed experiments on all three data sets, and the various update models, comparing SUP and fbSUP. We have measured both the improvement in effective utility of fbSUP over SUP and the additional cost in terms of number of probes. Our results show that for a variety of traces fbSUP can improve the effective utility by up to 110% compared to SUP with only moderate increases in the number of probes (less than 50% increase).

Figure 5 provides a comparison of SUP and fbSUP for the RSS data set with life=`overwrite`. Figure 5(a) presents the increase in relative utility with fbSUP for four variations of FPN. For $Z = 1.0$, SUP performance is optimal and therefore fbSUP cannot improve the schedule. For smaller FPN values, fbSUP does not improve performance for $X = 1$ since its logic converges to that of SUP for this notification rule. For larger X values, however, fbSUP improves significantly (for this data set, up to 104% for $X = 5$ and $Z = 0.8$).

The cost of fbSUP is presented in Figure 5(b). Again, for $Z = 1.0$ no modification to the schedule is needed and fbSUP adds no extra probes. For other models, and for $X > 1$, effective utility improvement comes at a cost, albeit not a big one. Therefore, for $Z = 0.4$ and $X = 5$, the increase in the number of probes was 49% (compare with 68% increase in effective utility). It is noteworthy that the increase in effective utility and probing is not necessarily correlated. For example, for $X = 4$, the effective utility of fbSUP for $Z = 0.6$ is dropping, while the number of probes slightly increase.

6 Related work

Existing pull-based data delivery approaches can be classified along several dimensions. The first dimension is the *objective* of the problem. The objective is the utility to be optimized; by utility we mean some client-specified function to measure the value of an object to a client, based on a metric such as data recency [1] or importance to the client [3]. The second dimension is the *constraints* of the problem. Constraints are restrictions, e.g., bandwidth, to which the model should adhere. A third dimension is *when objects are refreshed*, either in the background, on-demand when clients request them, or some combination of the two. In this section we describe several existing pull-based approaches, both background and on-demand, with different objectives and constraints.

Several approaches assume no resource constraints. TTL [8] is commonly used to maintain freshness of object copies for applications such as on-demand Web access. Each object is assigned a Time-to-Live and any object requested after this time must be validated at a server to check for updates. The objective of TTL is to maximize the recency of the data. Latency- recency profiles [1] are a generalization of TTL that allow clients to explicitly trade off data recency to reduce latency using a utility function. The objective is to maximize the utility of all client requests, assuming no bandwidth constraints.

RSS feeds [11] provide an interface for Web sources to publish updates to clients. RSS readers convert pull-based Web sources to push by periodically polling sources in the background to check for updates. RSS provides no explicit bandwidth constraints, but readers typically monitor sources at fixed intervals (e.g., every 15 minutes), which may either consume excessive bandwidth or fail to meet client requirements.

There are also many approaches that aim to maximize an objective function (e.g., recency, utility), subject to explicit resource constraints. These approaches assume that the contraints are given a priori. TTL with pre-validation [Pre-Validation] [5] extends TTL by validating expired cached objects in the background. This approach assumes limits on the amount of bandwidth for pre-validation, but assumes no bandwidth constraints for on-demand requests. WIC [9] monitors updates to a set of information sources in the background subject to bandwidth constraints. The objective is to capture updates to a set of objects. WIC (and also CAM [10]) monitor updates to a collection of sources, but it assumes bandwidth constraints are given and does not consider client utility per-se, although user preferences (referred to as *urgency*) are included in the utility computation.

Cache synchronization [4] and application-aware cache synchronization [AA-Synch] [2] refresh objects in the background to maximize the average recency of a set of objects in a cache, subject to bandwidth constraints. These approaches maximize average recency of a set of objects rather than monitoring updates to sources, and unlike SUP do not consider client utility.

Profile-driven cache management [3] is an on-demand approach that enables data recharging for clients with intermittent connectivity. Clients specify complex profiles of the utility of each object. The objective is to download a set of objects to maximize client utility while the client is connected, thus, there is an explicit limit on the number of objects that can be downloaded. We note that PDCM does not directly handle updates to objects.

A key observation is that the above approaches maxi-

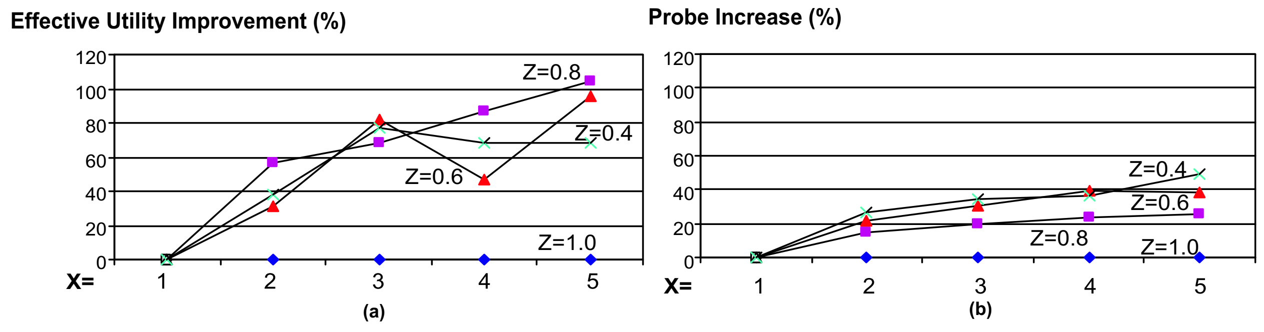

Figure 5. Relative performance of SUP and fbSUP for RSS data, life=overwrite

mize an objective function, subject to either unlimited resources (e.g., bandwidth) or explicit resource constraints. None of the above approaches explicitly minimize resource consumption subject to satisfying a target recency or utility. Existing background approaches such as WIC, CAM, and Cache Synchronization, as well as PDCM [3] assume that bandwidth constraints are given and do not attempt to use fewer probes than the pre-specified constraint value. Thus, existing approaches may have many wasteful probes. In contrast, in this paper our proposed algorithm SUP solves the dual problem of minimizing the number of probes to sources subject to utility constraints. Unlike existing approaches, SUP monitors sources using a minimal number of probes.

7 Conclusions

Pull-based data delivery is needed for many applications to support diverse profiles across multiple sources, preserve user privacy, and reduce resource consumption. Minimizing the number of probes to sources is important for pull-based applications to conserve resources and improve scalability. Solutions that can adapt to changes in source behavior are also important due to the difficulty of predicting when updates occur. In this paper we have addressed these challenges with two algorithms, SUP and fbSUP, that aim to satisfy user profiles and minimize resource consumption. We have formally shown that SUP is optimal for $OptMon_2$ and under certain restrictions can be optimal for $OptMon_1$ as well. We have empirically shown, using data traces from diverse Web sources that SUP can satisfy user profiles and capture more updates compared to existing policies. We have also analyzed the impact of profiles, life parameters, and update models on SUP online performance. fbSUP was introduced to increase the utility of SUP by dynamically changing monitoring schedules. Our experiments show that fbSUP improves on SUP with a moderate increase in the number of needed probes.

In future work, we will consider how to incorporate resource constraints into SUP. We will also consider source monitoring for mobile applications which introduces new challenges due to intermittent connectivity. Finally, we plan to investigate using profiles for grid and network monitoring applications.

References

[1] L. Bright and L. Raschid. Using latency-recency profiles for data delivery on the web. *Proc. Very Large Data Bases*, 2002.

[2] D. Carney, S. Lee, and S. Zdonik. Scalable application-aware data freshening. *Proc. ICDE*, 2003.

[3] M. Cherniack, E. Galvez, M. Franklin, and S. Zdonik. Profile-Driven Cache Management. *Proceedings of 19th International Conference on Data Engineering (ICDE)*, 2003.

[4] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. *Proc. ACM SIGMOD Conf.*, 2000.

[5] E. Cohen and H. Kaplan. Refreshment Policies for Web Content Caches. *Proceedings of IEEE INFOCOM*, 2001.

[6] P. Deolasee, A. Katkar, P. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. *Proc. 10th WWW Conf.*, 2001.

[7] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: a stochastic modeling approach. *Journal of the ACM*, 48(6):1141–1183, 2001.

[8] J. Gwertzman and M. Seltzer. World wide web cache consistency. *Proc. USENIX Technical Conference*, 1996.

[9] S. Pandey, K. Dhamdhere, and C. Olston. Wic: A general purpose algorithm for monitoring web information sources. *Proceedings of Very Large Databases (VLDB)*, 2004.

[10] S. Pandey, K. Ramamritham, and S. Chakrabarti. Monitoring the dynamic web to respond to continuous queries. In *Proceedings of the International World Wide Web Conference (WWW)*, pages 659–668, May 2003.

[11] RSS. *http://www.rss-specifications.com*.

[12] R. Wolski, L. Miller, O. Graziano, and M. Swany. *Performance Information Services for Computational Grids*. Kluwer, 2004.