Stochastic Consistency, and Scalable Pull-Based Caching for Erratic Data Sources

ABSTRACT

We introduce the notion of stochastic consistency, and propose a novel approach to achieving it for caches of highly erratic data. Stochastic consistency guarantees that errors in cached values remain within a user-specified bound, with a user-specified probability. Erratic data sources, such as stock prices, sensor data, and network traffic statistics, are common and important in practice. However their erratic patterns of change can make caching hard. We use a Brownian motion model to capture the behaviour of data changes, and use the underlying theory to predict when caches should initiate pulls to refresh cached copies to maintain stochastic consistency. Our approach allows servers to remain totally stateless, thus achieving excellent scalability. We also discuss a new real-time scheduling approach for servicing pull requests at the server. Our scheduler delivers prompt response whenever possible, and minimizes the aggregate cache-source deviation due to delays during server overload. We conduct extensive experiments to validate our model on real-life datasets, and show that our scheme outperforms current schemes. We also show that our scheduler achieves higher scalability and lower cache-source deviation than FCFS scheduling, which is used widely.

1. INTRODUCTION

Caching is widely used to reduce latency, client-server bandwidth, and server load, but it remains very difficult to guarantee cache consistency with erratic and volatile data. Many applications must confront the challenge of efficiently delivering erratically changing data to a large population of clients. We propose a novel approach to this problem.

Important examples of erratic data sources include sensor streams [1], stock market data [2], and network statistics [3]. Such sources produce continuous but highly fluctuating data, making it hard to predict their behaviour. Nonetheless, such erratic data sources are commonly encountered in domains such as the WWW, data warehousing, and streaming applications [4]. For example, a stock broker may have hundreds of thousands of clients, but still be required to notify clients when the value of a stock or a portfolio changes by some predetermined value. Similarly, in a sensor query processing system [1], a large network of sensors may continuously stream data into a central query processor, with users having registered continuous queries with the query processor. Efficiency in delivering query results to users in accordance with their precision requirements is crucial to query processor's scalability and responsiveness.

1.1 Caching and Content Delivery

Since it is impractical to query data sources constantly, caching is commonly used to address such problems, which generally fall under the rubric of content delivery [5]. Caches for content delivery are managed using either the *push* or *pull* model. The push method [6, 7, 8, 9] disseminates updates to all proxies whenever data changes at the server. While push can achieve strict consistency, it suffers from two drawbacks. First, it is not scalable, since it requires considerable resources, including CPU power, sockets, memory space, to maintain connections with its proxies. Second, it is less reliable, since state information about connections with proxies is lost when the server fails, and is hard to restore upon reboot.

The pull method [10, 11, 12], in contrast, lets proxies decide when to refresh local copies. The server can be stateless, so that the method is scalable and resilient to server failure. However, the effectiveness of a pull scheme depends largely on the proxy's ability to initiate pulls at the proper times. Combinations of push and pull have also been tried [13, 14]. The success of such combined schemes can also depend on the pull strategy at the proxy.

1.2 Stochastic Consistency for Erratic Data

We propose a novel pull-based synchronization scheme for maintaining *stochastic consistency* (see Section 4) of frequently and unpredictably changing (erratic) data, applicable when users are willing to tolerate some error.

Our notion of stochastic consistency guarantees that cache-source deviation remains within user-specified error tolerance with a certain probability level. In many applications, slightly out-of-sync values are satisfactory, if they are within specified error bounds. For example, a stock holder may want to track stock price changes higher than $0.1, and a system administrator may only care when machine loads change by more than 10%.

In such cases, strict consistency between cache and source is unnecessary. It suffices to adaptively synchronize cached copies with the source guided by user-defined error toler-

ances. Also, because of the erratic nature of such data, it is desirable to associate a confidence metric with the cache-source error. For example, a stock holder may be satisfied with a quote if it is within \$0.1 of its true value, with confidence 90%. In this paper, we show how to achieve stochastic consistency by modeling the evolution of erratic data as Brownian motions.

The success of pull schemes depends largely on effective modeling of source data evolution. Some recent work examines stochastic modeling of web page content evolution. In [15, 16], the authors examine various synchronization policies, assuming that each web page is modified by a Poisson process with a certain rate. A formal discussion of stochastic modeling of content evolution in relational database appears in [17]. The authors use compound nonhomogeneous Poisson processes to model the behavior of record insertion and deletion, and Markov chains to model attribute modification. However, none of these models are suitable for erratic data with user provided error tolerance.

1.2.1 Our Contributions

We make several contributions in this paper. First, we introduce the notion of stochastic consistency, and demonstrate that it is a reasonable consistency model for many practically important classes of erratic data.

Second, we show how to model source object evolution as Brownian motions. We verify many real-life datasets can be modeled as Brownian motions by experiments. Proxies can schedule pulls adaptively, using this model to determine when the expected error in the cached copy exceeds user-specified error tolerance. We also show how proxies update their Brownian motion models adaptively, in accordance with changes in source data characteristics. Although we present a pure pull scheme, it can be applied seamlessly to other push-and-pull schemes, say, as in [14].

Third, we solve a novel real-time scheduling problem for processing pull requests at the server. When the number of proxies is large, the server may be overwhelmed by bursts of pull requests, resulting in poor response. Our scheduling method aims to minimize cache-source disparities caused by such delays overall, and to gracefully degrade system performance. As far as we are aware, this problem formulation has not appeared in the literature.

Finally, we study the performance of our approach to real-life data by experiments. We examine stock traces, system load data collected from university servers and real-time sensor data sampled from distributed ocean buoys, showing that all of them can achieve good *fidelity* (see Section 5). We compare the performance of our Brownian motion based pull scheme with the *Adaptive TTL Scheme* proposed in [11]. We also simulate our scheduling algorithm and compare it with the simple FCFS scheme.

The rest of this paper is organized as follows: section 2 reviews some previous work on consistency models and data synchronization techniques. We briefly review the Brownian motion model in section 3, and verify our datasets conform to the model well. In section 4, we discuss how to apply Brownian motion to maintain stochastic consistency, and give the adaptive pull algorithm. Simulation results are given in section 5. We discuss how to schedule pull requests at the server to minimize overall cache-source deviation in section 6. Section 7 concludes this work.

2. RELATED WORK

Data synchronization in replicated systems has been widely studied. Since strong consistency incurs high overheads and poor scalability, weak consistency is frequently preferred [18, 19, 20, 7]. Techniques such as lazy replication [18], epsilon-serializability [19], and anti-entropy [21] have been proposed.

Alonso et al. [7] introduce the concept of *quasi-caching*, which allows the cached value to deviate from the true value in a controlled way. Several coherency conditions are proposed, including the *delay* condition, which states by how much time a cached image may lag its central object, and the *arithmetic* condition, which gives the allowable difference between the values of the object and cached image. Several synchronization techniques are proposed, among which *implicit invalidation* is a pull-based technique, which forces refreshing by invalidating the cached copy after a certain time. However the authors have not discussed how to set the invalidation time, based on the update pattern of the central object. The concept of *probabilistic consistency* is introduced in [22], which guarantees that the value returned by the system is temporally consistent with the newest copy with a probability p. However, this approach does not consider the important related problem of bounding the errors in cached values.

Maintaining temporal consistency of erratic, frequently changing (dynamic) data is studied in [11, 14, 8, 23]. In [8], pure push is used to disseminate updates through a tree of cooperating repositories. In [11], various schemes are discussed for clients to calculate the time to refresh cached copies, so that temporal coherency is maintained. If $U(t)$ and $S(t)$ denote the cached and source values at time t, respectively, and c is the desired error bound, then the goal is to maintain the constraint $|U(t) - S(t)| < c$. They experimentally show that the *adaptive TTL scheme* has the best temporal consistency properties among various TTL schemes proposed in the paper. However, the adaptive TTL scheme uses a simple linear model to model the evolution of a erratic source, which fails to capture the frequent fluctuations associated with the data. In Section 5, we show by experiments that our Brownian motion based pull scheme outperforms *adaptive TTL scheme*. Deolasee et al. [14] propose an adaptive push-pull algorithm PaP, which combines push and pull. In the algorithm, the performance of the client side pull is very crucial to the overall performance. Yu et al. [23] study bounding numerical errors among replicated servers, where every server can store and accept updates. In their approaches, each server has to keep state information of other servers, which is not possible for our system with potentially large number of proxies.

Temporal consistency issue has also been discussed in the web domain [24, 25, 26, 15]. The work in [16] verifies the modeling of web page updates as Poisson process, by experiments on more than half a million web pages over four months. Cho et al. [15] study how to synchronize local copies of web data with their remote sources. Poisson processes are adopted to model the content evolution of web data, and several synchronization policies are discussed and compared. Erratic sources are typically numeric data, and changes to them are much more frequent than those due to web page evolution. We need a new model for such erratic data, which can dynamically capture source data characteristics.

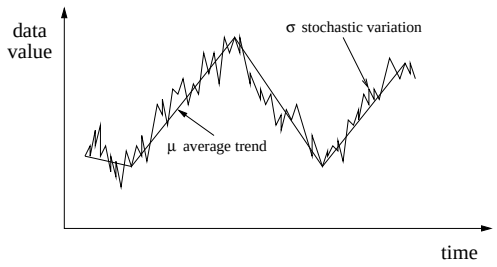


Figure 1: Brownian motion with drift

Symbol	Company Name	β (Volatility)
NOVL	Novell	2.33
BRCM	Broadcom	3.91
SEBL	Siebel Systems	3.06
YHOO	Yahoo Inc	3.88
QCOM	Qualcomm	2.05
SUNW	Sun Micro	2.23
CSCO	Cisco	1.98
XLNX	Xilinx	2.07
GE	General Electric	1.1

Table 1: Stock traces used in our simulation

3. USING BROWNIAN MOTION MODELS

The Brownian motion [27], is widely used to model fluctuating data in finance, engineering, communications, physics, and so on. It models increments in random data as independent Gaussian samples. We first describe the model and demonstrate that it models many practical erratic datasets well.

3.1 The Brownian Motion Model

A stochastic process W_t is called a *Standard Brownian motion (SBM)* [27] if it satisfies three conditions: (1) $W_0 = 0$, (2) $W_t - W_s$ is normally distributed with mean 0 and variance $t - s$, and (3) $W_t - W_s$ is independent of $W_v - W_u$ if (s, t) and (u, v) are non-overlapping time intervals. Property (2) says every increment of SBM is normal deviate. In general, SBM is a Martingale process [27], meaning loosely that the best estimate for its future value is its current value.

Drifting Brownian Motion (DBM) is similar, but includes a secular drift in the expectation of the process (see Figure 1). Its behavior is captured by the difference equation

$$\Delta S_t \ = \ \mu_t \Delta t + \sigma_t \Delta W_t, \qquad (1)$$

or the stochastic differential equation $dS_t = \mu_t dt + \sigma_t dW_t$. μ_t and σ_t are called *drift* and *diffusion* parameter, respectively, both of which are time dependent parameters. Fundamentally, DBM is a combination of a predictable linear trend and a Brownian motion process. The term $\mu_t \Delta t$ represents the non-stochastic part of the process, and characterizes the current moving trend. The term $\sigma_t \Delta W_t$ is the stochastic or Brownian motion part, and represents the randomness in the data. At time t, the process increment ΔS_t follows the normal distribution $(\mu_t \Delta t, \ \sigma_t^2 \Delta t)$.

3.2 Applicability of Brownian Motion

A key property of Brownian motions is that data increments are modeled as independent normal distributions. In practice, the drift and diffusion parameters μ and σ may change over the time. Consequently, we should only expect μ and σ to be constant in the short term. To show that this model is useful in caching, it suffices to show that increments of non-overlapping, equal-length time intervals are samples from normal distributions in the short term.

Normality testing [28, 29] has been extensively studied in the statistical literature, because of the great importance of the normal distribution. Various tests exist [28, 29], including the Kolmogorov-Smirnov (*K-S*) test, the Chi-Square (χ^2) test, the Wilk-Shapiro (*W*) test, and the Anderson-Darling (*A-D*) test. Some of these are general *goodness-of-fit* tests, such as the *K-S* test and χ^2 test, while others are specifically designed for testing normality, such as the *W* test. Generally speaking, the tests specific to normality are more powerful in detecting non-normality than the general goodness-of-fit tests [28]. We chose to apply the *W* test for testing normality, as it has high *power* [28], given no prior knowledge about the possible alternatives.

We tested the applicability of the Brownian motion model to a variety of real-life streaming data sources. We selected datasets from three classes of real-life data, namely, stock prices, sensor data, and system load data, and tested whether their increments were normal. The stock streams we chose are listed in Table 1 with values arriving every minute for the entire year 06/2001–06/2002. Each stream contained about 10^5 data values. Our sensor time series were taken from the TAO project [30] at the Pacific Marine Environmental Laboratory (PMEL), and comprised a year's measurements (11/1991–11/1992) of temperature (*temp*) at various sea levels. Each *temp* stream contained about 10^4 values, sampled every 1 minute. Our system load data comprised 1-minute averages of system loads collected every five seconds for two days on our main server.

We deliberately chose data streams with high volatility. Such data show high uncertainty of movement, and display large fluctuations even over short intervals. Highly erratic data are more challenging for our adaptive pull model. The β value shown in Table 1 is a measure of the relative volatility of a stock to the market. Generally, symbols with $\beta \in [1, 4]$ are considered to have high volatility.

Table 2 shows the average *p-values* [28] of the *W* test for various time intervals. We form the *null hypothesis* that the samples are normal. The p-value measures the probability that the test statistic (*W*) will take on a value that is at least as extreme as the observed value when the null hypothesis is true. In our context, the larger the p-value, the stronger the confidence with which we may accept the samples as normal. The *significance level* (α) of our test is 0.05. Any sample with p-value lower than α can be flagged as non-normal with high confidence. The p-values for our datasets are far larger than α, indicating that we can have high confidence in modeling the increments as normal samples. Not surprisingly, for longer time intervals, the p-value drops somewhat, suggesting the model may evolve during longer intervals (see Section 4.3).

4. STOCHASTIC CONSISTENCY AND ITS REALIZATION

Figure 2 depicts our system model. In our approach, proxies cache and serve data objects under a stochastic consistency model. A user is satisfied if the value returned by the

	stock traces			*temp* traces			system load
	BRCM	*QCOM*	*SEBL*	*0N/140W/36M*	*0N/140W/47M*	*0N/140W/70M*	trace
10 min	75.50%	80.28%	75.50%	75.21%	72.96%	79.58%	76.00%
15 min	71.80%	75.67%	76.17%	72.38%	75.90%	79.44%	75.23%
20 min	72.14%	70.88%	76.11%	73.45%	73.55%	77.20%	75.41%
30 min	70.92%	65.23%	72.14%	71.47%	66.13%	62.21%	70.59%

Table 2: Average *p-values* of W test for various datasets and time intervals, confidence interval: 95%. *0N/140W/36M* trace is sampled at longitude 0N, latitude 140W, sea depth 36M.

proxy for object o_i is within ϵ of its true value with probability at least p. Let $S_i(t)$ and $C_i(t)$ be the true and cached values of object o_i at time t. Let ϵ be the user-specified error tolerance and p be the confidence expressed as a probability. The cache is stochastically consistent if

$$\Pr\,[\,|(S_i(t) - C_i(t))| \leq \epsilon\,] \geq p, \quad \text{at all times } t. \qquad (2)$$

A proxy must frequently refresh cached copies to maintain stochastic consistency. To reduce communication overhead and server loads, we need a mechanism to adaptively decide when the cache-source deviation is likely to exceed ϵ, and refresh the cached copy only at such times.

Let the proxy have pulled object o_i's values from the server at times $t_1, t_2, \ldots, t_k$. At time t_k, the proxy must determine the next time $t_{k+1} = t_k + \Delta t_k$ the data must be pulled. During the interval $[t_k, t_{k+1}]$, the proxy returns to the user an estimate for o_i's value, based on the last pulled value $S_i(t_k)$. Consider the probability function:

$$F_i(t_k, \Delta t_k) = \Pr[|S_i(t_k + \Delta t_k) - E(S_i(t_k + \Delta t_k))| \leq \epsilon] \quad (3)$$

$F_i(t_k, \Delta t_k)$ is the probability that the cache-source deviation of o_i's value is within ϵ at time $t_k + \Delta t_k$. $S_i(t_k + \Delta t_k)$ is the actual source value of object o_i at that time, and $E(S_i(t_k + \Delta t_k))$ is the value estimated at the proxy. How the proxy determines $E(S_i(t_k + \Delta t_k))$ depends on the source data evolution model, which will be discussed shortly.

The cached value is stochastically consistent at time $t_k + \Delta t_k$ if $F_i(t_k, \Delta t_k) \geq p$. Clearly, to maintain stochastic consistency, the proxy must pull to refresh the local value before $F_i(t_k, \Delta t_k)$ drops below p. Thus, at time t_k, the next pull time for object o_i is determined by the smallest Δt_k for which $F_i(t_k, \Delta t_k) \leq p$.

In what follows, we describe our adaptive pull system in Section 4.1. In Section 4.2, we show how to determine the next pull time under the Brownian motion model, so that the user-specified error tolerance and the probability confidence are met. The *drift* and *diffusion* parameters, which capture data characteristics on the fly, are estimated for each source data object periodically at the server. The issue of parameter estimation is discussed in Section 4.3. In 4.4, we describe the adaptive pull algorithm.

4.1 System Architecture

There are three major components in our system (see Figure 2): a central server or erratic data source, proxies with caches (only one proxy is shown), and clients. The server maintains N data objects $\{o_1, o_2, \ldots, o_N\}$, whose values can change independently. Users request object values through proxies. At each proxy, the user registers tuple (o_i, ϵ_j, p_j) specifying that the user is interested in object o_i, and can tolerate error ϵ_j with probability confidence p_j. Different users may, in fact, specify different values of ϵ_j and p_j for the same o_i. Proxies adaptively pull object values on behalf of users, at intervals designed to maintain stochastic consistency with the server. The server responses have the form $(S_i(t_p), \mu_i(t_p), \sigma_i(t_p))$, where $S_i(t_p)$ denotes the value of o_i at pull time t_p, $\mu_i(t_p)$ and $\sigma_i(t_p)$ are the estimates of o_i's drift and diffusion parameters, respectively (see Section 4.3).

When a user requests the data object from the proxy at some time t, the current estimate of o_i's value, $E(S_i(t))$, is returned. In our scheme, proxies can do better than simply return the last pulled value $S_i(t_p)$, which is the default approach in most previous work. This ability is due to the fact that our Brownian motion model (see Section 4.2) is capable of capturing source data evolution in some detail.

As pull requests arrive at the server from proxies, a real-time scheduler dynamically schedules these requests for service (see Section 6), to ensure that responses are prompt.

4.2 Determining the Next Pull Time

We model the evolution of each source object as a drifting Brownian motion. We justified this approach in Section 3.2. Let $\Delta S_i(t, \Delta t) = S_i(t + \Delta t) - S_i(t)$, object o_i's increment $\Delta S_i(t, \Delta t)$ can be modeled by the equation:

$$\Delta S_i(t, \Delta t) = \mu_i(t) \cdot \Delta t + \sigma_i(t) \cdot \Delta W_t \qquad (4)$$

$\mu_i(t)$ and $\sigma_i(t)$ are the time varying drift and diffusion parameters, respectively. W_t is a standard Brownian motion. It's not difficult to see that $\Delta S_i(t, \Delta t)$ follows a normal distribution:

$$\Delta S_i(t, \Delta t) \sim N(\mu_i(t)\Delta t,\ \sigma_i^2(t)\Delta t) \qquad (5)$$

Let $E(S_i(t + \Delta t)) = S_i(t) + \mu_i(t)\Delta t$, we obtain:

$$S_i(t + \Delta t) - E(S_i(t + \Delta t)) \sim N(0,\ \sigma_i^2(t)\Delta t) \qquad (6)$$

In Equation 6, $E(S_i(t + \Delta t))$ is the expected value of $S_i(t + \Delta t)$ at time $t + \Delta t$. Thus $E(S_i(t + \Delta t))$ can serve as the best estimation of the actual value $S_i(t + \Delta t)$ at such time. Suppose $S_i(t)$ is pulled by the proxy at time t, $E(S_i(t+\Delta t))$ will be returned to users upon requests before $S_i(t)$ expires. Let the error be $I_i(t, \Delta t) = S_i(t + \Delta t) - E(S_i(t + \Delta t))$, we also need to find such Δt_ϵ so that the probability that the error $I_i(t, \Delta t_\epsilon)$ is within ϵ at time $t + \Delta t_\epsilon$ drops below p.

Equations 3 and 6 indicate that $F_i(t, \Delta t)$ is a decreasing function of Δt. Thus, it suffices to find Δt_ϵ for which $\Pr[|I_i(t, \Delta t_\epsilon)| \leq \epsilon] = p$. One must solve the following equation to obtain Δt_ϵ:

$$\int_{-\epsilon}^{\epsilon} \frac{1}{\sqrt{2\pi \cdot \sigma_i(t)^2 \cdot \Delta t}} exp(-\frac{x^2}{2\sigma_i(t)^2 \cdot \Delta t})dx = p, \qquad (7)$$

Despite its imposing look, Equation 7 is a very routine calculation on normal distributions. By solving this equation,

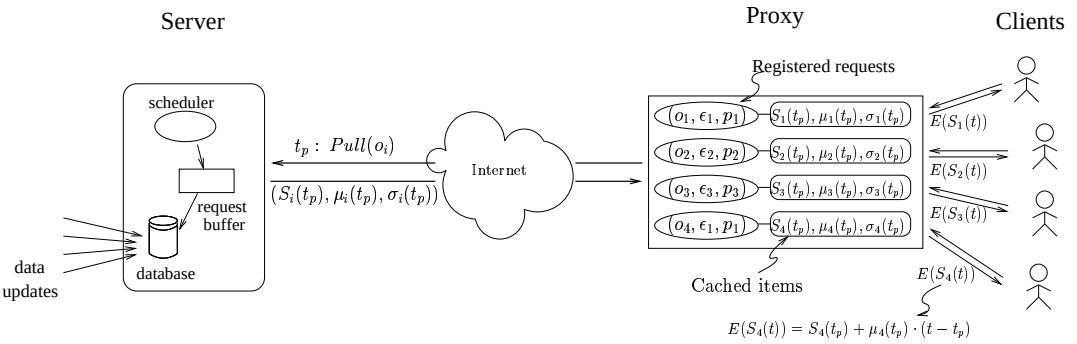


Figure 2: System architecture

proxies can determine the next time, $t + \Delta t_\epsilon$ to refresh the local copy so that the expected error won't exceed user specified tolerance.

We have thus far treated error tolerances as absolute values. However, one might be also interested in relative error tolerance. For example, a user may need to know the value of a stock portfolio within a given percentage error tolerance. If we denote the relative error tolerance as $\epsilon_r\%$, at time t, we need to find $\Delta t_{\epsilon_r\%}$ such that:

$$\Pr[|\frac{S_i(t+\Delta t) - E(S_i(t+\Delta t))}{S_i(t)}| \leq \epsilon_r\%] = p \qquad (8)$$

or

$$\Pr[|S_i(t+\Delta t) - E(S_i(t+\Delta t))| \leq \epsilon_r\% \cdot |S_i(t)|] = p \qquad (9)$$

Now we can similarly calculate the next pull time as the case of absolute error tolerance described before.

4.3 Updating the Model

Proxies need the current drift parameter $\mu_i(t)$ to calculate the expected object value $E(S_i(t))$, and the diffusion parameter $\sigma_i(t)$ to calculate Δt_ϵ. Both parameters reflect the current characteristics of o_i, and should be estimated on a regular basis at the server.

The server maintains a buffer B_i containing the k most recent data values for each object o_i, sampled at regular intervals, h. When a new update of o_i comes in to a full buffer, the oldest value in the buffer is simply discarded. Parameters $\mu_i(t)$ and $\sigma_i(t)$ are estimated using the contents of B_i at time t. The issue of estimating these parameters has been extensively studied, and maximum-likelihood estimators are typically used [27]. We show in our experiments, that for the short time horizons we will be concerned with, simpler methods work quite well.

To estimate the DBM's drift parameter $\mu_i(t)$, we use the sample mean over the elements in B_i. Let $\hat{\mu}_i(t)$ represent the estimated value of $\mu_i(t)$, we have:

$$\hat{\mu}_i(t) = \frac{\sum_{j=0}^{k-2}(B_i[j+1] - B_i[j])}{(k-1)\cdot h} \qquad (10)$$

To estimate the DBM's diffusion parameter $\sigma_i(t)$, we calculate the sample variance of the increments of o_i.

$$\hat{\sigma_i}^2(t) = \frac{1}{(k-2)\cdot h}\sum_{j=0}^{k-2}(B_i[j+1] - B_i[j] - \hat{\mu}_i(t))^2 \qquad (11)$$

Algorithm 1 Adaptive Pull Algorithm

```
Proxy side (o_i, ε, p):
loop
  if t_curr == t_pull(o_i) then
    /* time to pull new value of object o_i */
    Send a pull request pull(o_i) to the server, and wait;
    On receiving server response, proxy obtains (S_i, μ_i, σ_i);
    Update local copy with new value S_i;
    Calculate Δt_ε based on obtained μ_i and σ_i;
    t_last(o_i) = t_pull(o_i),   t_pull(o_i) = t_curr + Δt_ε;
  end if

  if proxy receive a user request req(o_i) then
    /* the proxy receives a user request for object o_i */
    E(t_curr) = S_i + μ_i(t_curr − t_last);
    Return E(t_curr) to the user;
  end if
end loop

Server side:
loop
  if server receive a pull request pull(o_i) then
    Process the request, retrieve current value of object o_i, S_i;
    Retrieve latest evaluated μ_i and σ_i;
    Send (S_i, μ_i, σ_i) back to the proxy;
  end if
end loop
```

$\hat{\mu}_i(t)$ and $\hat{\sigma_i}^2(t)$ are both unbiased and easy to compute. Since h is comparatively small, the estimated values are quite accurate. The buffer size k is adjusted experimentally to achieve the best performance.

The parameter estimation process for each object must be carried out repeatedly at the server. On processing a pull request, the latest estimates of μ_i and σ_i are returned with the object values to the proxy (see Figure 2), which will use them to calculate expected object value and decide the next pull time.

4.4 Adaptive Pull Algorithm

Algorithm 1 outlines our adaptive pull algorithm. t_{curr} is the current system time at the proxy. $t_{pull}(o_i)$ is the calculated next pull time for object o_i, and $t_{last}(o_i)$ is the last time the cached copy of o_i is updated. On receiving response (S_i, μ_i, σ_i) from the server, the proxy calculates Δt_ϵ based on σ_i as well as ϵ and p. On receiving user request at time t_{curr}, the proxy returns its best estimated value $E(t_{curr})$ based on current cached value S_i and μ_i. Our server is to-

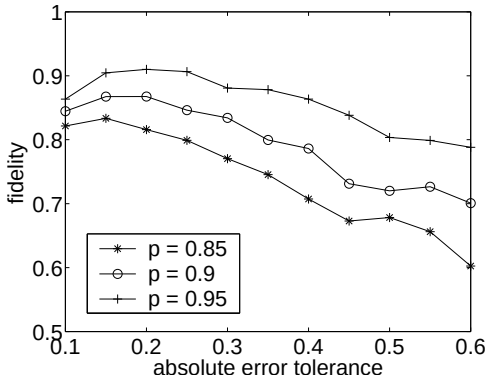


Figure 3: Fidelity for stock trace BRCM

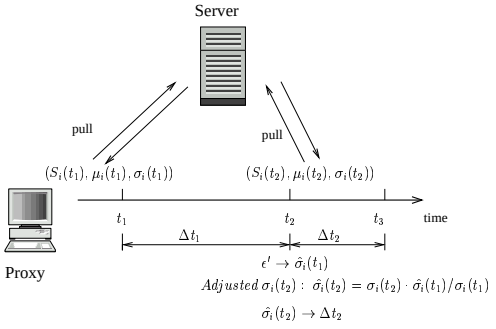


Figure 4: On-line adjustment of σ

tally stateless, guaranteeing good scalability. For simplicity, we assume the pulled value is immediately available at the proxy after it sends the pull request, which may not be true in the real world. We will tackle this problem in section 6.

It is quite straightforward to extend our model to hierarchical proxying schemes. As user requests (o_i, ϵ, p) propagate up the hierarchy, each non-leaf node picks the most conservative value of ϵ and p for each object cached in the subtree, and propagate them upwards. Root nodes communicate with the server using the adaptive pull scheme we proposed. Updates are pushed down the hierarchy according to ϵ value at each node.

5. ADAPTIVE PULL PERFORMANCE

We conducted a series of experiments to demonstrate the performance of our scheme on real-life erratic data sources, such as stock prices, sensor data, and system loads. We showed in Section 3.2 that it is appropriate to model such datasets as Brownian motions. Here we show that our approach outperforms previous approaches. We simulated our approach using the `csim` simulation package [32] on an Intel Pentium 4 at 1.70GHz.

5.1 Adjusting σ On-line

As in Section 3.2, we deliberately chose highly erratic datasets, since they represent greater challenges for our method than do ordinary datasets. In particular, we must confront a paradox in the case of highly erratic data—performance actually degrades as we increase error tolerance, due to rapid second-order changes in data characteristics. Values fluctuate rapidly in erratic datasets, causing σ_i to be large, but a second-order behavior is that the values of σ_i and μ_i are themselves likely to change rapidly. This effect leads to poor predictions of pull times, since the data evolution model will likely have changed significantly even before the current Δt_ϵ has expired. The *fidelity* metric [11] measures how often our predictions of pull times meet the user-specified error tolerance. Fidelity characterizes the confidence we may place in our model, and is defined as:

$$fidelity(o_i) = \frac{\text{time cache-source errors are } \leq \epsilon\ (\epsilon_r\%)}{\text{total simulation time}}$$

Figure 3 illustrates this paradox on the trace for the highly volatile stock BRCM, since fidelity drops off as we increase the error tolerance. This stock has the extremely high *beta* value of 3.91, so that its σ is also likely to change rapidly. As we increase the error tolerance, the interval between pulls increases. Unfortunately, the value of σ is likely outdated quite early in this interval, so that our predictions of next pull time is likely to be wrong. On the other hand, evaluating σ often is a bad option.

5.1.1 A Heuristic for Evaluating the Effective σ

Our approach is to correct *observed* values for σ_i to obtain *effective* values for σ_i, based on the observed error in our predictions. Since Δt_ϵ is determined by σ_i alone (see Equation 7), these corrected values lead to better predictions of pull times. In effect, our method uses second-order changes in σ_i for one interval to predict second-order changes in the next one.

At time t_1, the proxy pulls values $(S_i(t_1), \mu_i(t_1), \sigma_i(t_1))$ for object o_i from the server, and calculates Δt_1 based on $\sigma_i(t_1)$. At time $t_2 = t_1 + \Delta t_1$, we expect the error to be within ϵ with probability p. Now, at time t_2, let the proxy pull $(S_i(t_2), \mu_i(t_2), \sigma_i(t_2))$, and let the actual error be $\epsilon' = |S_i(t_2) - S_i(t_1) - \mu_i(t_1)\Delta t_1|$. If $\epsilon' > \epsilon$, our predictions based on $\sigma_i(t_1)$ were likely to have been incorrect due to changes in σ_i during Δt_1. We compute the *effective* σ_i value $\hat{\sigma}_i(t_1)$ that would have resulted in error ϵ' at time t_2 as follows:

$$\int_{-\epsilon'}^{\epsilon'} \frac{1}{\sqrt{2\pi \cdot \hat{\sigma}_i(t_1)^2 \cdot \Delta t_1}} exp(-\frac{x^2}{2\hat{\sigma}_i(t_1)^2 \cdot \Delta t_1}) dx = p$$

Since $\epsilon' \geq \epsilon$, $\hat{\sigma}_i(t_1) \geq \sigma_i(t_1)$. To predict the next pull interval, we first compute the ratio $\hat{\sigma}_i(t_1)/\sigma_i(t_1)$, and correct the value of $\sigma_i(t_2)$ returned by the server as follows to obtain an adjusted $\sigma_i(t_2)$:

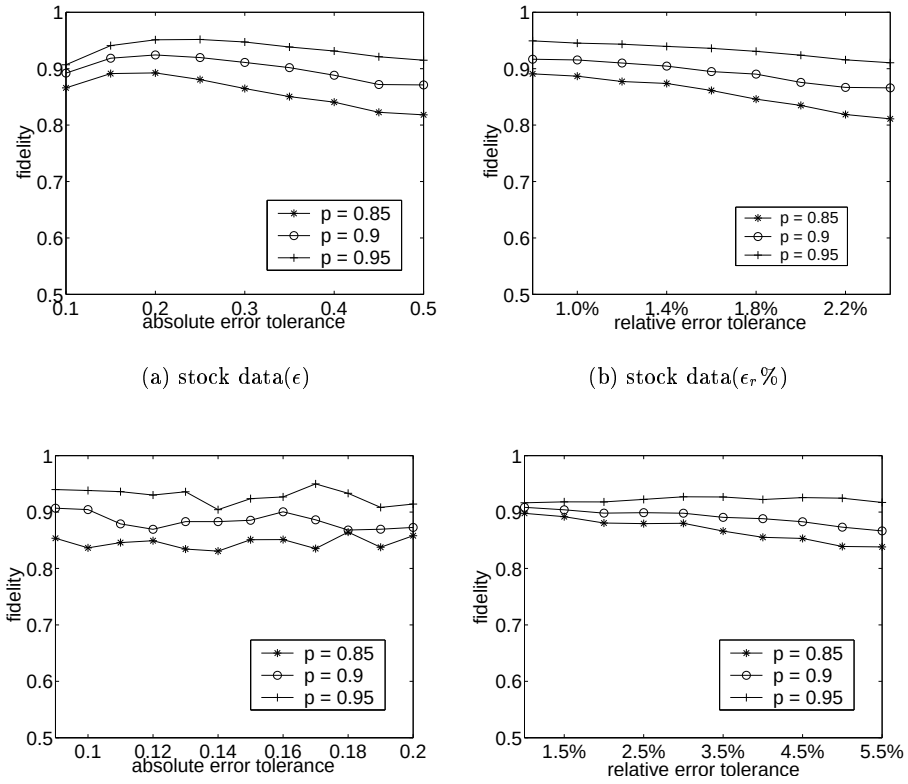
$$\hat{\sigma}_i(t_2) = \sigma_i(t_2)\frac{\hat{\sigma}_i(t_1)}{\sigma_i(t_1)}$$

We now calculate Δt_2 using $\hat{\sigma}_i(t_2)$ instead of $\sigma_i(t_2)$ at time t_2. This heuristic is illustrated in Figure 4.

5.2 Performance of Our Approach

Equipped with the σ adjustment heursitic discussed above, we first demonstrate the *fidelity* of our approach on the three datasets with both absolute error tolerance (ϵ) and relative error tolerance ($\epsilon_r\%$). Due to space limitations, we only show part of our results in Figure 5.

Each point in Figure 5 is the averaged fidelity over five time series. The fidelity achieved by our model closely matches the prespecified confidence p. For example, the average fidelity at 90% level is 89.8% for stock data(ϵ), 89.4% for stock



(a) stock data(ϵ) (b) stock data(ϵ_r%)

(c) *temp* sensor data (ϵ) (d) system load data (ϵ_r%)

Figure 5: Fidelity

data (ϵ_r%), 88.4% for *temp* data (ϵ), and 89.1% for system load data (ϵ_r%). Our results suggest that our approach captures source data change patterns well, and predicts the next pull time well. Not surprisingly, a higher p value achieves higher fidelity, but incurs larger communication overhead, as explained below.

Figure 6 demonstrates the communication overhead incurred for different datasets. The communication overhead is measured as the number of pulls generated during the entire simulation period, and varies across the datasets, since it depends on the dataset's trends and fluctuation patterns. A higher confidence level p triggers more pulls, while a looser error bound triggers fewer pulls. The *total updates* curve shows the total number of data updates in the time series. The *optimal* curve shows the minimum number of pulls needed for a certain error tolerance, obtained from an off-line calculation. These two curves represent the number of pulls for a naive scheme and the optimal scheme respectively.

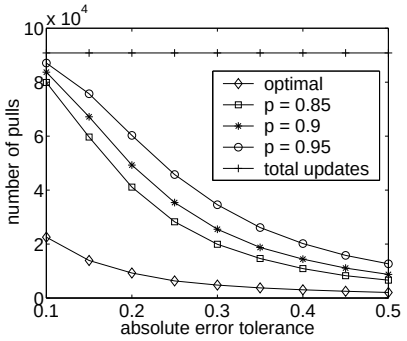
5.3 Comparison with Adaptive TTL Scheme

A linear adaptive pull scheme (*Adaptive TTL*) with user-provided temporal coherency requirement was introduced in [11, 14]. Their notion of *temporal coherency requirement* is equivalent to our *error tolerance*. *TTL* (time-to-live) denotes the time interval before the current local copy is invalidated, i.e, the time before next pull. The scheme applies a linear change model on source objects. Proxies estimate the linear coefficients based on their last retrieved values and the latest calculated TTL (T_l). The next TTL is determined by a weighted combination of estimated TTL (T_{est}), the latest TTL, and the most conservative TTL thus far (T_{mr}). [14],

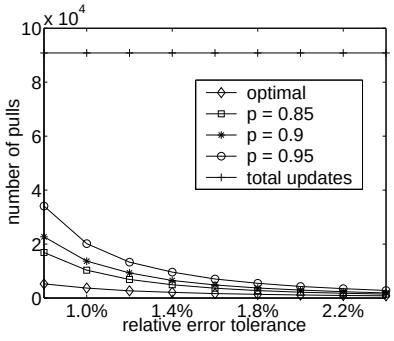
$$T = \max(T_{min},\ \min(T_{max},\ a \cdot T_{mr} + (1-a) \cdot T_{dyn}))$$

where $T_{dyn} = w \cdot T_{est} + (1-w) \cdot T_l$, $T_{est} = (T_l / |D_{latest} - D_{penultimate}|) \cdot \epsilon$, and $[T_{min}, T_{max}]$ denotes a static bound for the next TTL value. In [11], the authors compare *Adaptive TTL* with other pull schemes by experiments, and conclude that *Adaptive TTL* outperforms all other schemes.

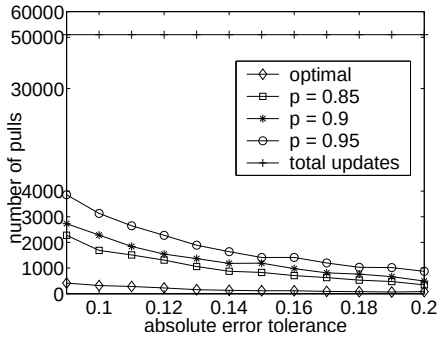
We compare our scheme with *Adaptive TTL* on various stock traces, in Figure 7. An ideal scheme is such one that can achieve high fidelity with minimum number of pulls. We record the number of pulls needed by each scheme to achieve the same fidelity, and show the results for three individual stock traces (BRCM, SEBL, and QCOM). The parameters for *Adaptive TTL* are as follows: $w = 0.5$, $TTR_{min} = 0.3$, $TTR_{max} = 500$. The coefficient a is dynamically adjusted to match the fidelity achieved by our approach. Clearly, for all traces we show, to achieve the same fidelity, *Adaptive TTL* scheme requires far more pulls than our scheme, which suggests that our Brownian motion model can capture the source data characteristics much better than *Adaptive TTL*'s linear model.



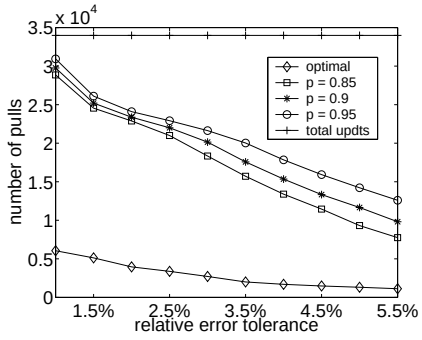
(a) stock data: QCOM (ϵ)



(b) stock data: SEBL (ϵ_r%)



(c) *temp* data: *0N/140W/36M* (ϵ)



(d) system load data (ϵ_r%)

Figure 6: Number of pulls

6. PULL REQUEST SCHEDULING AT SERVER

Having demonstrated how well our Brownian motion-based adaptive pull model works with real-life examples of erratic data, we now show how to use this model in practice. Server scalability is one of our major goals, since we want to maximize the number of proxies that a server can handle. This is a difficult goal since data characteristics may change over time. If σ_i becomes too high, the request rate for o_i at the server may be high (see Equation 7) as well as bursty. Proxies may then experience long delays, or even receive no replies at all, if the server queue overflows, causing errors in cached copies to exceed user-specified tolerance. In the spirit of the pull model, we would also like the server to remain stateless. We show how to attain these goals using a variant of *Just-In-Time (JIT)* scheduling.

In the adaptive strategy we have described in Section 4, pulls are carefully scheduled to strike a balance between user-specified error tolerance and confidence requirements, and reduce computation and communication overhead at the server and proxies. Paradoxically, this makes it as bad for the server to respond too soon to requests as it is to respond too late, since it upsets this careful balance we have tried to achieve.

Our scheduling problem at the server is a novel variant of *JIT* scheduling, in which a task must complete before its due time, *but as close to it as possible*. We present a scheduler that guarantees prompt responses whenever possible, and minimizes the total error across the proxies when the server is overloaded. The form of the penalty function we address (see Section 6.2), as far as we know, has not been previously addressed.

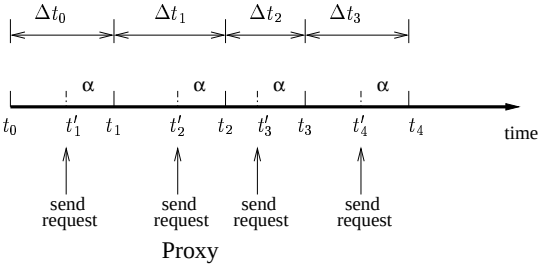
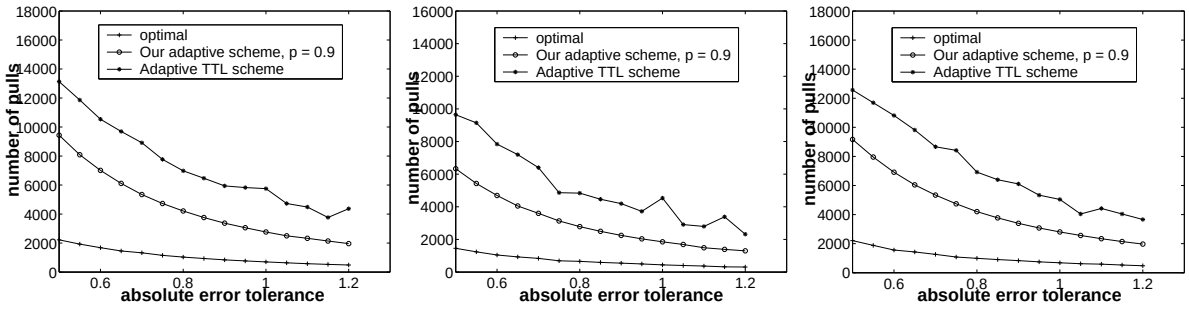


Figure 8: Our scheduling model

6.1 Scheduling Model Options

Figure 8 shows the scheduling of pull requests for object o_i from the proxy. $t_1, t_2, \cdots, t_k$ are the times the proxy is supposed to refresh the cached value. Missing these times may cause the cache-source deviation exceeds the user tolerance.

In a naive scheme, the proxy would send a request to the



(a) BRCM trace (b) SEBL trace (c) QCOM trace

Figure 7: Comparing our approach with *Adaptive TTL* scheme on stock traces.

server at time $t_{k+1} = t_k + \Delta t_k$, the point when the data is needed. The server processes incoming requests on FCFS basis. However, this strategy gives the server very little leeway, since it must respond hastily to each request.

Another approach would be for the proxy to notify the server as soon as it computes Δt_k, so that the server has the maximum latitude in scheduling requests. However, as noted before, the server can not respond too soon, and must effectively deliver the current value of o_i, μ_i and σ_i at time t_{k+1}. The server must keep track of the Δt_k for a potentially large number of proxies, and can no longer be called stateless.

Our strategy represents a good compromise. Proxies issue pull requests α time units before they are really needed, so that the the k-th pull request is made at time $t_{k+1} - \alpha$ (see Figure 8). The scheduler at the server now has more flexibility in deciding when to process each request, and it retains no proxy-specific state. We show how to derive α in section 6.4.

Since our scheduling algorithm needs to know the last drift parameter returned to the proxy (see Section 6.2), a pull request for o_i has the form $pull(o_i, \mu_i')$, where μ_i' is the last value of μ_i provided by the proxy. Request processing at the server may involve retrieval of S_i, σ_i, and μ_i from the database, assembling HTML pages or XML documents, encrypting data, and so on. In fact, different proxies may interact with different server applications, so even for the same o_i, requests from different proxies may require different processing times. For simplicity, we assume constant network delays. Using more complex models of network delays will be our future work.

6.2 Real-Time Scheduling Model

Each pull request, $pull(o_i, \mu_i')$, arriving at the server can be treated a real-time task, parametrized as $T_j(o_i, \mu_i', \mu_i, \sigma_i, p_j, d_j)$, where μ_i' is the proxy's last value of drift parameter; μ_i and σ_i are the current drift and diffusion parameters, respectively; p_j is T_j's processing time; d_j is its due time, $d_j = t + \alpha - \delta$, ($t$ is server's current time, and δ is the network delay. Since the request is sent at time $t_{k+1} - \alpha$, only $\alpha - \delta$ time units are available for server processing.)

Since object values change over time, we must process the request as close as possible to the due time d_j, that is, at $d_j - p_j$, to provide the proxy the latest object value. Such scheduling is called *Just-In-Time (JIT)* scheduling [33, 34, 35], and penalizes earliness, as well as tardiness. Tasks finish exactly on their due dates is an ideal *JIT* schedule.

Since scheduling aims to minimize the overall cache-source deviation, deviations of the request processing time from $d_j - p_j$ incur penalties. If the proxy has not received the update by time t_{k+1}, it will continue to use the old drift parameter μ_i' to estimate the object's value. Meanwhile, the actual drift and diffusion parameters at source are μ_i and σ_i. Thus, at any future time $t_{k+1} + \Delta t$, the cache-source value deviation will consist of two parts: a drift-induced component, $|\mu_i - \mu_i'|\Delta t$, and a diffusion-induced component, $\sigma_i\sqrt{\Delta t}$ (see Equation 6).

Now, we can define a penalty function incorporating both parts, to estimate how far a returned data value deviates from its *JIT* value. If C_j is the actual completion time of T_j, the penalty is defined as:

$$P_j = \tilde{\mu_i} \cdot |C_j - d_j| + \sigma_i \cdot \sqrt{|C_j - d_j|} \qquad (12)$$

where $\tilde{\mu_i} = |\mu_i - \mu_i'|$. We use the absolute value of the deviations, so that the penalty is zero only when $C_j = d_j$. Our goal is to minimize $\sum_j P_j$, the total penalties.

JIT scheduling has been well studied, and Baker et al. [33] review the literature on scheduling n tasks on single machine to minimize total earliness and tardiness penalty. Most work to date uses linear penalty functions, such as $w_j \max(0, d_j - C_j) + h_j \max(0, C_j - d_j)$, where w_j is the early cost rate, and h_j is the tardy cost rate. It is known that minimizing an aggregate linear penalty function is NP-hard [34]. Since our version (Equation 12) is a combination of a linear penalty and a square root penalty, it is more general, and clearly, also NP-hard. We seek heuristics to solve our problem.

6.3 An Off-Line LINSQT-ET Heuristic

Our scheduling algorithm must work efficiently and online, since scheduling decisions have to be made as pull requests arrive. We start with the heuristic *LIN-ET* proposed for off-line scheduling with linear penalties [34].

THEOREM 1. *Given n tasks, let the objective be to minimize $\sum_{i=1}^{n}(w_i \cdot |C_i - d_i|)$. All adjacent pairs of tasks in the optimal sequence must satisfy*

$$w_i p_j - 2\Omega_{ij} w_i \geq w_j p_i - 2\Omega_{ji} w_j,$$

where task i immediately precedes task j, and Ω_{ij} and Ω_{ji} are defined as:

$$\Omega_{xy} = \begin{cases} 0 & if\ s_x \leq 0, \\ s_x & if\ 0 < s_x < p_y, \\ p_y & otherwise. \end{cases} \quad (13)$$

Here $s_x = d_x - t - p_x$ is the slack of task x, p_x is its processing time, and t is the earliest time the machine is free.

Proof: See [34]. □

In Theorem 1, the early cost rate and the tardy cost rate are treated symmetrically. The theorem offers a necessary condition for an optimal schedule, from which the *LIN-ET* heuristic [34] is derived as follows: given n tasks at time t, they are sequenced in order of priority R_i, calculated as follows:

$$R_i(s_i) = \begin{cases} W_i & if\ s_x \leq 0, \\ W_i - 2W_i s_i / k\overline{p} & if\ 0 < s_x < k\overline{p}, \\ -W_i & if\ s_x \geq k\overline{p}. \end{cases} \quad (14)$$

where $W_i = w_i/p_i$, $\overline{p}$ is the average processing time of n tasks, and k is called *lookahead parameter*, used to extend the scope of optimality beyond two adjacent tasks. In practice, as discussed in [34], a low k ($k = 2$, or $k = 3$) may be adequate.

LIN-ET performs quite well under various settings [34], and is efficient, so it is a good candidate for an on-line heuristic. Before designing the heuristic for our problem, which we will call *LINSQT-ET*, we must first reconcile our non-linear penalty function (Equation 12) and the priority representation (Equation 14) as follows.

THEOREM 2. *Given n tasks, let the objective be to minimize $\sum_{i=1}^{n}(\tilde{\mu}_i \cdot |C_i - d_i| + \sigma_i \cdot \sqrt{|C_i - d_i|})$. All adjacent pairs of tasks in the optimal sequence must satisfy*

$$w_i p_j - 2\Omega_{ij} w_i \geq w_j p_i - 2\Omega_{ji} w_j$$

where

$$w_i = \tilde{\mu}_i + \frac{\sigma_i}{\sqrt{|s_i|} + \sqrt{|s_i - p_j|}} \quad (15)$$

The other notations are as in Theorem 1.

Proof: See Appendix. □

From theorem 2, our scheduling problem can be mapped to linear penalty case (Theorem 1) with weight w_i defined in Equation 15. Thus, our heuristic *LINSQT-ET* sequences tasks in order of priority R_i defined in 14, using w_i as in Equation 15.

6.4 An On-line LINSQT-ET Heuristic

The heuristic LINSQT-ET we have just described is off-line, but Algorithm 2 outlines an on-line version.

According to Equation 14, tasks whose slack times exceed $k\overline{p}$ have the lowest priorities ($-W_i$), and are unlikely to be scheduled in the immediate future. In other words, if a task has slack time $k\overline{p}$ when it arrives, there is little danger of it having arrived later. Consequently, if δ is the network delay, a proxy needs to initiate a pull no earlier than $k\overline{p} + p_i + 2\delta$ time units ahead of its due time. So we can derive $\alpha = k\overline{p} + p_i + 2\delta$ in Figure 8.

In Algorithm 2, pending tasks are ordered by their priorities in *PRIO-LIST*. Task priority is a function of the slack time (Equation 15), and changes with time. However, updating priority values and reordering *PRIO-LIST* requires

Algorithm 2 On-line LINSQT-ET heuristic

PRIO-LIST: List of tasks with $s_j < k\overline{p}$, in priority order.

On Arrival of Task T_j:
/ calculate priority of T_j: W_j is as in Equation 15 */*
$T_j \cdot priority = W_j - 2W_j s_j / k\overline{p}$;
insert T_j into *PRIO-LIST*;

Task Execution:
/ when one task done, pick next task in PRIO-LIST */*
loop
  if *PRIO-LIST* is not empty **then**
    update priorities of top h tasks in *PRIO-LIST*, and reorder them;
    T_k = *PRIO-LIST*(0);
    dequeue T_k from *PRIO-LIST*, and execute;
  else
    / if PRIO-LIST is empty, wait for time ω. */*
    wait(ω);
  end if

  for every l time units **do**
    / update the whole PRIO-LIST regularly */*
    update all priorities in *PRIO-LIST*;
  end for
end loop

$O(n \log n)$ time. This is too expensive to perform before every scheduling decision, especially when n is large. Instead, we schedule after updating and reordering only the top h tasks in *PRIO-LIST*. However, we do update and reorder all of *PRIO-LIST* every l time units. In our experiments, we set $h = 10$ and $l = 10$ *sec*.

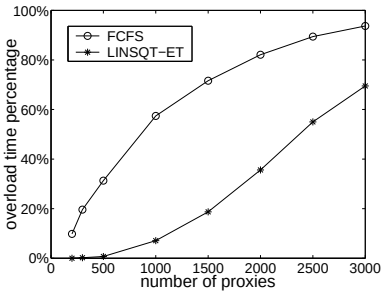
6.5 Experimental Results

We simulate our *LINSQT-ET* scheduling algorithm on stock traces, and compare its performance with the FCFS-based scheduling under various workloads. FCFS scheduling is used in current methods [36]. We set the constant lookahead parameter $k = 3$. To calculate α, proxies obtain the value of k, $\overline{p}$, and p_i the first time they contact the server. As mentioned earlier, requests from different proxies may require different processing times. The request processing time for each proxy is randomly drawn from the range 25ms–40ms [37]. Figure 9 compares FCFS and *LINSQT-ET* with respect to two metrics: the fraction of time the server is overloaded, and the average penalties during overload. As Figure 9 (a) shows, the server gets more overloaded as the number of proxies increases in both schemes. However, *LINSQT-ET* remains far more scalable than FCFS throughtout the wide range considered. Besides, as Figure 9(b) shows, *LINSQT-ET* scheduling incurs significantly less penalty, and consequently less cache-source deviation than FCFS.

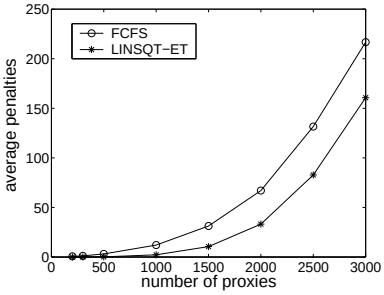
Figure 10 shows the penalty incurred by our heuristic method relative to the optimal, computed through an off-line exhaustive search. The deviation does increase with the number of pending tasks, but is still relatively low.

7. CONCLUSIONS

We have proposed *stochastic consistency*, a new model of consistency appropriate for situations when users can tolerate some error. We have also presented a novel pull-based scheme for maintaining stochastic consistency for caches holding erratic and volatile data. Our approach models changes



(a) fraction of time server is overloaded



(b) average penalties incurred during overload

Figure 9: Performance comparison between FCFS and LINSQT-ET heuristic ($p = 0.95$, $\epsilon \in [1.0, 2.2]$)

in source data as Brownian motions, and schedules pulls from the proxy to keep errors in the cached data within user-specified error tolerance. Pulls are initiated at times determined by user error tolerance, probability confidence and data characteristics.

Servers remain fully stateless in our method, so that our approach is very scalable. To guarantee high scalability and prevent the server from becoming a bottleneck, the server schedules pull requests using a new variant of *JIT* scheduling. Our variant uses a non-linear penalty function, which has not been addressed previously in the literature.

We show through simulations that our Brownian motion based scheme achieves high fidelity on stock traces, sensor data and system load data, while keeping the communication overhead low. We also compare our adaptive scheme with the *adaptive TTL* scheme in [11], and find that to achieve the same fidelity, our scheme requires much fewer pulls. We simulate the server scheduling algorithm under various workloads, and demonstrate that it far outperforms naive FCFS scheme both in scalability and overload penalties.

Acknowledgement

This work was supported in part by grants from Tata Consultancy Services, Inc., the Digital Media Innovations program of the University of California, and by the Fault-Tolerant Networks program of the Defense Advanced Research Projects Agency, under contract F30602-01-2-0536.

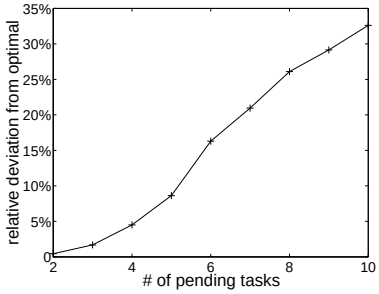


Figure 10: Our heuristic method vs optimal

APPENDIX

Proof of theorem 2: Given n tasks $\{T_1, T_2, \ldots, T_n\}$, the optimal sequence π^* minimizes the cost function $O_{\pi^*} = \sum_{i=1}^{n}(\mu_i \cdot |C_i - d_i| + \sigma_i \cdot \sqrt{|C_i - d_i|})$. For any pair of adjacent tasks T_i and T_j in π^*, interchanging their positions should not decrease O_{π^*}. Since interchanging two adjacent tasks doesn't affect the cost contributed by other tasks in sequence, we only need to care about the subsequence of T_i and T_j. We thus have $O(ij) \leq O(ji)$, where $O(ij)$ is the cost of subsequence (i, j), and $O(ji)$ is the cost of (j, i).

We first consider the case when T_i and T_j complete before their due times in both sequences. We thus have:

$$\mu_i(d_i - t - p_i) + \sigma_i\sqrt{(d_i - t - p_i)} + \mu_j(d_j - t - p_i - p_j) + \sigma_j\sqrt{(d_j - t - p_i - p_j)} \leq \mu_j(d_j - t - p_j) + \sigma_j \cdot \sqrt{(d_j - t - p_j)} + \mu_i(d_i - t - p_i - p_j) + \sigma_i\sqrt{(d_i - t - p_i - p_j)}$$

Here t is the completion time of last job before T_i and T_j. It's not difficult for us to derive:

$$(\mu_i + \frac{\sigma_i}{\sqrt{d_i - t - p_i} + \sqrt{d_i - t - p_i - p_j}})p_j \leq (\mu_j + \frac{\sigma_j}{\sqrt{d_j - t - p_j} + \sqrt{d_j - t - p_i - p_j}})p_i$$

Since $s_i = d_i - t - p_i$, $s_j = d_j - t - p_j$, and let $w_i = \mu_i + \frac{\sigma_i}{\sqrt{|s_i|}+\sqrt{|s_i - p_j|}}$ we have:

$$-w_i p_j \geq -w_j p_i$$

According to equation 13, since $s_i \geq p_j$, and $s_j \geq p_i$, we can derive:

$$w_i p_j - 2\Omega_{ij} w_i \geq w_j p_i - 2\Omega_{ji} w_j$$

Now consider the case when T_i and T_j both complete before due times in sequence (i, j), however T_i is late in sequence (j, i). We then have:

$$\mu_i(d_i - t - p_i) + \sigma_i\sqrt{(d_i - t - p_i)} + \mu_j(d_j - t - p_i - p_j) + \sigma_j\sqrt{(d_j - t - p_i - p_j)} \leq \mu_j(d_j - t - p_j) + \sigma_j \cdot \sqrt{(d_j - t - p_j)} + \mu_i(t + p_i + p_j - d_i) + \sigma_i\sqrt{(t + p_i + p_j - d_i)}$$

We can derive:

$$(\mu_i + \frac{\sigma_i}{\sqrt{s_i} + \sqrt{p_j - s_i}})(p_j - 2s_i) \geq (\mu_j + \frac{\sigma_j}{\sqrt{s_j} + \sqrt{s_j - p_i}})p_i$$

According to equation 13, since $s_i \leq p_j$, and $s_j \geq p_i$, we also have:

$$w_i p_j - 2\Omega_{ij} w_i \geq w_j p_i - 2\Omega_{ji} w_j$$

We can similarly derive the above inequality for the other cases. Thus Theorem 2 is proved. □

A. REFERENCES

[1] Samuel Madden and Michael J. Franklin, "Fjording the stream: An architecture for queries over streaming sensor data," in *Proc. of the 18th ICDE Conf*, San Jose, 2002.

[2] "http://www.traderbot.com," .

[3] Flip Korn, S. Muthukrishnan, and Yunyue Zhu, "Checks and balances: Monitoring data quality problems in network traffic databases," in *Proc. of the 29th VLDB Conf*, Berlin, Germany, 2003.

[4] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom, "Models and issues in data stream systems," in *Proc. 21th ACM SIGACT-SIGMOD-SIGART Symp. on Principle of Database Systems*, Madison, May 2002.

[5] Stefan Saroiu, Krishna P. Gummadi, Richard Dunn, Steven D. Gribble, and Henry M. Levy, "An analysis of internet content delivery systems," in *Proc. of the 5th OSDI Conf*, Boston, MA, December 2002.

[6] Swarup Acharya, Michael Franklin, and Stanley Zdonik, "Disseminating updates on broadcast disks," in *Proc. of the 22nd VLDB Conf*, Mumbai, India, September 1996.

[7] Rafael Alonso, Daniel Barbara, and Hector Garcia Molina, "Data caching issues in an information retrieval system," in *ACM Trans. Database Systems*, p. 15(3). September 1990.

[8] Shetal Shah, Krithi Ramamritham, and Prashant Shenoy, "Maintaining coherency of dynamic data in cooperating repositories," in *Procs. of the 28th VLDB Conf.*, Hong Kong, 2002.

[9] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Arun Lyengar, "Engineering server-driven consistency for large scale dynamic web services," in *Proc. of the 10th WWW Conf*, Hong Kong, May 2001.

[10] J. Gettys, J. Mogul, et al., "Hypertext transfer protocol – http/1.1," in *RFC 2616*. 1999.

[11] Raghav Srinivasan, Chao Liang, and Krithi Ramamritham, "Maintaining temporal coherency of virtual data warehouses," in *The 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[12] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari, "Adaptive leases: A strong consistency mechanism for the world wide web," in *The 19th INFOCOM Conf.*, Tel Aviv, Israel, March 2000.

[13] Swarup Acharya, Michael Franklin, and Stanley Zdonik, "Balancing push and pull for data broadcast," in *Proc. of the 1997 ACM-SIGMOD Conf*, Tucson, May 1997.

[14] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy, "Adaptive push-pull: Disseminating dynamic web data," in *The 10th WWW Conference*, Hong Kong, May 2001.

[15] Junghoo Cho and Hector Carcia Molina, "Synchronizing a database to improve freshness," in *Proc. of the 2000 ACM-SIGMOD conference*, Dallas, May 2000.

[16] Junghoo Cho and Hector Garcia Molina, "The evolution of the web and implications for an incremental crawler," in *Proc. of the 26th International Conference on Very Large Databases*, Cairo, Egypt, September 2000.

[17] Avigdor Gal and Jonathan Eckstein, "Managing periodically updated data in relational databases: A stochastic modeling approach.," in *Technical Report, Rutgers University Center for Operations Research*, 2001.

[18] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems*, vol. 10, no. 4, 1992.

[19] Calton Pu and Avraham Leff, "Replica control in distributed systems: An asynchronous approach," in *SIGMOD Conference*, 1991.

[20] Hengming Zou, Nandit Soparkar, and Farnam Jahanian, "Probabilistic data consistency for wide-area applications," in *Proc. of the 16th ICDE Conf*, San Diego, February 2000.

[21] Richard A. Golding, *Weak-Consistency Group Communication and Membership*, Ph.D. thesis, 1992.

[22] Chi Zhang and Zheng Zhang, "Trading replication consistency for performance and availability: an adaptive approach," in *Proc. of the 23rd ICDCS Conf*, Providence, Rhode Island, May 2003.

[23] Haifeng Yu and Amin Vahdat, "Efficient numerical error bounding for replicated network services," in *The VLDB Journal*, 2000.

[24] Ari Luotonen, *Web Proxy Servers*, Prentice Hall, 1998.

[25] Pablo Rodriguez, Christian Spanner, and Ernst W. Biersack, "Analysis of web caching architectures: Hierarchical and distributed caching," in *IEEE/ACM Transactions on Networking*, p. 9(4). August 2001.

[26] Pei Cao and Sandy Irani, "Cost-aware www proxy caching algorithms," in *Proc. of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997.

[27] S.Karlin and H.M.Taylor, *A First Course in Stochastic Processes, 2nd Edition*, Academic Press, 1975.

[28] Henry C. Thode, *Testing for Normality*, Marcel Dekker, Inc., 2002.

[29] Ralph B. D'Agostino and Michael A. Atephens, *Goodness-of-fit Techniques*, Marcel Dekker, Inc., 1986.

[30] "http://www.pmel.noaa.gov/tao/index.shtml," .

[31] Douglas C. Montgomery and George C. Runger, *Applied Statistics and Probability for Engineers*, John Wiley & Sons, Inc., 1999.

[32] "http://www.mesquite.com/htmls/guides.htm," .

[33] K.R.Baker and G.D.Scudder, "Sequencing with earliness and tardiness penalties: A review," in *Operations Research*, p. 38(1). 1990.

[34] P.S.Ow and T.E.Morton, "The single machine

early/tardy problem," in *Management Science*, p. 35(2). 1989.

[35] M.R.Garey, R.E.Tarjan, and G.T.Wilfong, "One-processor scheduling with symmetric earliness and tardiness penalties," in *Mathematics of Operations Research*, p. 13(2). 1988.

[36] "http://www.apache.org," .

[37] Jussara Almeida, Virgilio Almeida, and David Yates, "Measuring the behavior of a world-wide web server," Tech. Rep. 1996-025, 29, 1996.