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

Shanzhong Zhu Chinya V. Ravishankar
Department of Computer Science and Engineering
University of California, Riverside
Riverside, CA 92521
{szhu, ravi}@cs.ucr.edu

Abstract

We introduce the notion of stochastic consistency, and propose a novel approach to achieving it for caches of highly erratic data. Erratic data sources, such as stock prices, sensor data, are common and important in practice. However, their erratic patterns of change make caching hard. Stochastic consistency guarantees that errors in cached values of erratic data remain within a user-specified bound, with a user-specified probability. We use a Brownian motion model to capture the behavior of data changes, and use its 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 and reliability. 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.

1 Introduction

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

Erratic data are numerical data that change frequently and unpredictably, usually in response to uncontrollable environmental factors or other random influences in real systems. Erratic data have been recognized as common in streaming applications [11], and examples include sensor streams [23], stock prices [5], and network statistics.

Source-cache consistency for content delivery [27] is typically maintained under the *push* or the *pull* model. The push method [7, 10, 28, 33] monitors all data changes at the server, and disseminates updates to the client caches whenever data changes beyond user-specified error tolerance. Push is easy to implement, but suffers from several drawbacks.

First, push is not scalable, since the server must provide most of the required resources, such as processing power, sockets, and memory, and monitor data changes and manage communications with clients. Second, it is less reliable, since state information about connections with clients is lost when the server fails, and is hard to restore upon reboot. Finally, it is not power-efficient when clients are wireless and battery-powered, since clients must remain in listen mode, in anticipation of push updates.

1.1 The Pull Model

In the pull method [17, 19, 30], in contrast, clients decide when to refresh local copies. While the effectiveness of a pull scheme depends on the client's ability to initiate pulls at proper times, it is better than push in many real-world situations, especially with erratic and streaming data sources. Servers remains stateless, so the method is scalable and resilient to server failure. Wireless clients can sleep between scheduled pulls, saving power.

For example, in environmental monitoring systems, such as in [4], sensors are deployed at remote ocean lo-

*This work was supported by a grant from Tata Consultancy Services, Inc.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

cations at various depths to collect data. Such systems must be wireless and battery-powered because their locations may be hard to access. Research vessels are charged with collecting data updates with a certain accuracy, but frequently move out of wireless range of sensors to perform other functions. Since broadcasting takes much more power than to listen [6], it would be wasteful for the sensors to initiate data pushes, since there may be no vessels listening. We show in Section 4 how vessels can schedule visits to the area to pull data as appropriate to ensure monitoring accuracy constraints.

A similar problem arises when clients (such as PDAs or laptops) are power-limited. The push model would force the clients to remain in listen mode in anticipation of data. This is unacceptable, since the listen power is still significant, albeit less than in transmit mode. Our model allows clients to "sleep" most of the time, and change mode at pull times.

Finally, consider a system which manages stock portfolios for hundreds of thousands of clients, each with a local cache, for which the cache-source error must be less than some predetermined value. A model in which the server pushes updates to each client whenever the its cache error exceeds its threshold would simply not scale. The server would need to dedicate an enormous number of connections to this task.

Combinations of push and pull have also been proposed [8, 16] to achieve better performance. The success of such combined schemes also depends on the pull strategy at the client.

1.2 Stochastic Consistency for Erratic Data

We propose a novel pull-based synchronization scheme for maintaining *stochastic consistency* (see Section 4) of 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 tolerances. 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 [29].

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. Cho et al. [13, 14] verify the modeling of web page updates as Poisson process, by experiments on more than half a million web pages over four months, and use the model to synchronize local copies of web data with their remote sources. A formal discussion of stochastic modeling of content evolution in relational database appears in [18]. 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, since erratic sources are typically numeric data, and changes to them are much more frequent than those due to web page or relational database evolution. We need a new model for such erratic data, which can dynamically capture source data characteristics.

1.3 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 data evolution as Brownian motions. We verify many real-life datasets can be modeled as Brownian motions by experiments. Proxies, which cache data on behalf of clients, can schedule pulls adaptively, using this model to determine when the expected error in the cached copy exceeds user-specified error tolerance. Although we present a pure pull scheme, it can be applied seamlessly to other push-and-pull schemes, say, as in [16].

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. As far as we are aware, this problem formulation has not appeared in the literature before.

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 [30]. 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 Sec-

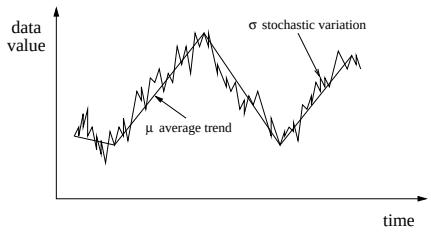


Figure 1: Brownian motion with drift

tion 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 paper.

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 [10, 22, 26]. Techniques such as lazy replication [22], epsilon-serializability [26], and anti-entropy [20] have been proposed.

Alonso et al. [10] 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 source value, and the *arithmetic* condition, which gives the allowable difference between the true values and cached image. A pull-based synchronization technique, *implicit invalidation*, is proposed, 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 source object.

The concept of *probabilistic consistency* is introduced in [37], which guarantees 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 issue of bounding the errors in cached values.

Maintaining temporal consistency of erratic, frequently changing (dynamic) data is studied in [16, 28, 30, 34]. In [28], pure push is used to disseminate updates through a tree of cooperating repositories. In [30], 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. [16] propose an adaptive *PaP* algorithm, which combines push and pull. In the algorithm, the performance of the client side pull is very crucial to the overall performance.

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

Table 1: Stock traces used in our simulation

Yu et al. [34] 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.

Wolfson et al. [32] study how to represent and update the locations of moving objects in spatial database. Their goal, which is to seek balance between location precisions and the update frequency, is similar to ours. Yet the nature of erratic data demands a different approach.

3 Using Brownian Motion Models

The Brownian motion [29], is widely used to model fluctuating data in finance, engineering, communications, physics, and so on. It models increments in random data as independent normal samples. We first describe the model and demonstrate that it models many practical erratic datasets well.

3.1 The Brownian Motion Model

A continuous-time stochastic process $W(t)$, which varies as a function of time t, is called a *Standard Brownian motion (SBM)* [29] if it satisfies three conditions: (1) $W(0) = 0$, (2) $W(t) - W(s)$ is normally distributed with mean 0 and variance $t - s$ $(t \geq s)$, 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 a normal deviate. In general, SBM is a Martingale process [29], meaning loosely that the best estimate for its future value is its current value.

Drifting Brownian Motion (DBM) $S(t)$ is similar, but includes a secular drift in the expectation of the process. Its behavior can be captured by the following equation:

$$dS(t) = \mu(t)\,dt + \sigma(t)\,dW(t), \qquad (1)$$

time interval	stock datasets			*temp* datasets			system load dataset
	BRCM	*QCOM*	*SEBL*	*0N/140W/36M*	*0N/140W/47M*	*0N/140W/70M*	
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-S* test for various datasets and time intervals, confidence interval: 95%. *0N/140W/36M* trace is sampled at longitude 0N, latitude 140W, sea depth 36M.

$\mu(t)$ and $\sigma(t)$ are the time-dependent *drift* and *diffusion* parameters, respectively. Using finite differences for differentials for simplicity, $\Delta W(t)$ represents the increment of the SBM, $\Delta W(t) \sim N(0, \Delta t)$. Intuitively, $\mu(t)$ models a secular upward or downward trend in the erratic data, while $\sigma(t)$ models the randomness associated with the data, as shown in Figure 1. 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 $\Delta S(t)$ follows the normal distribution $(\mu(t)\Delta t,\ \sigma^2(t)\Delta t)$.

3.2 Applicability of Brownian Motion

A key property of Brownian motions is that data increments are modeled as independent normal distributions $(\mu(t)\Delta t,\ \sigma^2(t)\Delta t)$. We expect the drift and diffusion parameters $\mu(t)$ and $\sigma(t)$ to be relatively constant in the short term. To show that this model is useful in modeling real-world datasets, it suffices to show that increments for small, non-overlapping, and equal-length time intervals are samples from normal distributions in the short term.

Normality testing [15, 31] has been extensively studied in the statistical literature, because of the great importance of the normal distribution. Various tests exist [15, 31], including the Kolmogorov-Smirnov (*K-S*) test, the Chi-Square (χ^2) test, the Wilk-Shapiro (*W-S*) test, and the Anderson-Darling (*A-D*) test. Each normality test is formulated as a *hypothesis test* in which the *null hypothesis* is that samples are normal. Some of these tests are general *goodness-of-fit* tests, such as the *K-S* test and χ^2 tests, while others are specifically designed for testing normality, such as the *W-S* test. Generally speaking, the tests specific to normality are more powerful in detecting non-normality than the general goodness-of-fit tests [31]. We chose to apply the *W-S* test for testing normality, as it has high *power* [31], 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 [4] at the Pacific Marine Environmental Laboratory (PMEL), and comprised a year's measurements (11/1991–11/1992) of temperature (*temp*) at various ocean depths. 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 [36] 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* [31] of the *W-S* test evaluated on increment samples taken over various time intervals. For each time interval, we repeatedly applied the *W-S* test on samples over the intervals through the entire data series, and record the average results (p-values). The p-value measures the probability that the *W-S* test statistic 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 higher 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 5.1).

4 The Stochastic Consistency Model

Figure 2 depicts our system model. There are three major components in our system: 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 are updated frequently, say, by incoming update streams. Users request object values through proxies. At each

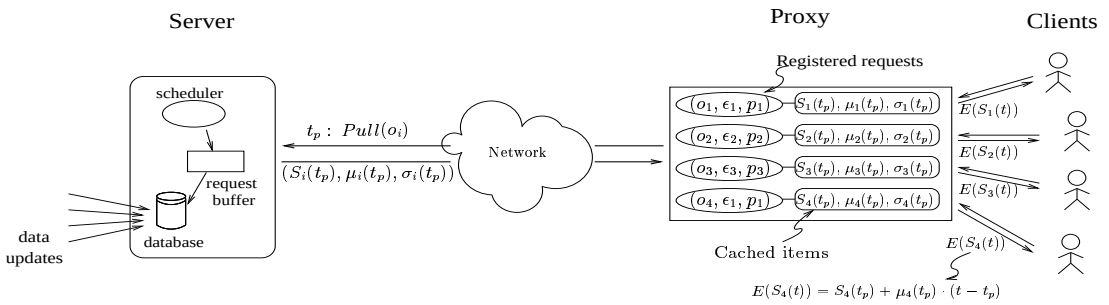


Figure 2: System architecture

proxy, users register tuples (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. 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 o_i's value from the proxy at some time t, the current estimate of o_i's value, based on last pulled object value and parameters, is returned.

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

In what follows, we introduce the stochastic consistency model in Section 4.1. In Section 4.2, we show how proxies can determine their pull times under the DBM model, to maintain stochastic consistency. The *drift* and *diffusion* parameters, which capture data characteristics on the fly, are estimated for each data object periodically at the server. The issue of parameter estimation is discussed in Section 4.3. In 4.4, we describe our adaptive pull algorithm.

4.1 Stochastic Consistency

In our approach, proxies cache and serve data objects under a stochastic consistency model. A client is satisfied if the value returned by the proxy for object o_i is within ϵ of its true value with probability at least p. Let $S_i(t)$ and $U_i(t)$ 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

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

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:

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

$F_i(t, \Delta t)$ is the probability that the cache-source deviation of o_i's value is within ϵ at time $t + \Delta t$, given the last proxy-pulled value is $S_i(t)$. $S_i(t+\Delta t)$ is the actual source value, and $E[S_i(t + \Delta t)]$ is the estimated value at the proxy. How the proxy determines $E[S_i(t + \Delta t)]$ 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 $t_{k+1} = t_k + \Delta t_k$ for object o_i is determined by the smallest Δt_k for which $F_i(t_k, \Delta t_k) \leq p$.

4.2 Determining the Next Pull Time

After justifying the appropriateness of the DBM model in Section 3.2, we model the increment of source object o_i, $\Delta S_i(t, \Delta t) = S_i(t + \Delta t) - S_i(t)$, as follows:

$$\Delta S_i(t, \Delta t) = \mu_i(t) \cdot \Delta t + \sigma_i(t) \cdot \Delta W(t), \quad (4)$$

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

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

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

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

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 estimate of $S_i(t + \Delta t)$ at such time. Suppose $S_i(t)$ and $\mu_i(t)$ are 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. We also need to find such Δt_ϵ that the probability the cache-source error remains within ϵ at time $t + \Delta t_\epsilon$, as defined in Equation 3, starts dropping 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:

$$\frac{1}{\sqrt{2\pi}} \int_{-\hat{\epsilon}}^{\hat{\epsilon}} \exp(-\frac{x^2}{2}) dx = p, \quad (7)$$

where $\hat{\epsilon} = \frac{\epsilon}{\sigma_i(t)\sqrt{\Delta t_\epsilon}}$. We note that this integral is simply the well-known error function [1], and since it is evaluated at the proxies and not at the server, our model remains scalable. Using Equation 7, proxies can obtain $t + \Delta t_\epsilon$, the next due time to refresh the local copy so that the expected error remains within 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 bound. If we denote the relative error tolerance as $\epsilon_r\%$, we need to find Δt_{ϵ_r} such that:

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

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 sampled value 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 current contents of B_i. According to the DBM model, increments follow normal distribution $N(\mu_i(t)\Delta t, \sigma_i^2(t)\Delta t)$. Given a fixed Δt, such as h, over short term, estimating $\mu_i(t)$ and $\sigma_i(t)$ is equivalent to estimating the mean and variance of increments over Δt. The issue of estimating mean and variance has been extensively studied, and sample mean and sample variance are typically used as estimators [29]. We show in our experiments, that for the short time horizons we will be concerned with, simpler methods work quite well. Let $\hat{\mu_i}(t)$ represent the estimated value of $\mu_i(t)$, we have:

$$\hat{\mu_i}(t) = \frac{1}{(k-1)h} \sum_{j=0}^{k-2} (B_i[j+1] - B_i[j]) \quad (9)$$

The estimated value of $\sigma_i(t)$ is:

$$\hat{\sigma_i}^2(t) = \frac{1}{(k-2)h} \sum_{j=0}^{k-2} (B_i[j+1] - B_i[j] - \hat{\mu_i}(t)h)^2 \quad (10)$$

$\hat{\mu_i}(t)$ and $\hat{\sigma_i}^2(t)$ are both unbiased and easy to compute. Since h is comparatively small, the estimated values are quite accurate.

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
```

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 totally 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 should be possible 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 values of ϵ and p for each object cached in the subtree, and propagate them upwards. Root nodes communicate with the server using our pull scheme. 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

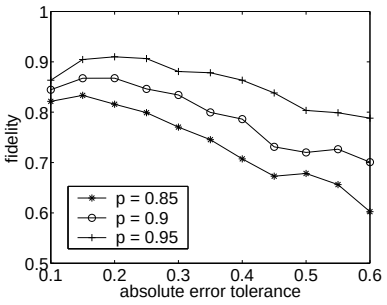


Figure 3: Fidelity for stock trace BRCM

sources, such as stock prices, sensor data, and system loads. We showed in Section 3.2 that it is appropriate to model these datasets as Brownian motions. Here we show that our approach outperforms previous approaches in terms of accuracy and efficiency. We simulated our approach using the `csim` simulation package [3] on an Intel Pentium 4 at 1.70GHz.

We first introduce the *fidelity* metric [30], which 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:

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

5.1 Adjusting σ_i 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 such highly erratic data—the *fidelity* of our approach actually degrades as we increase the error tolerance, due to rapid changes in data characteristics. Not only is σ_i high in this case, but 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.

Figure 3 illustrates this paradox on the highly erratic stock trace BRCM, since its fidelity drops off as we increase the error tolerance. BRCM has the extremely high β value of 3.91 (see Table 1), so its σ_i is also likely to change rapidly. As we increase the error tolerance, the interval between pulls increases. Unfortunately, the proxies can only obtain the updates of σ_i at those pull times, and the value of σ_i is likely outdated quite early in this interval, so that our predictions of next pull time is likely to be wrong.

The changing behaviour of σ_i, also known as *stochastic volatility*, has been extensively studied in Econometrics [12]. Typically, various stochastic process models are applied to $\sigma(t)$. We choose the *Hull-White (H-W)* model [12], since it is one of the simplest and very widely used. H-W models $\sigma_i^2(t)$ as a geometric Brownian motion [29]:

$$d\sigma_i^2(t) = \alpha(t)\sigma_i^2(t)\,dt + \beta(t)\sigma_i^2(t)\,dW_\sigma(t),$$

where $\alpha(t)$ and $\beta(t)$ are both time dependent coefficients, $W_\sigma(t)$ is a SBM uncorrelated with the $W(t)$ driving $S_i(t)$ (see Equation 4).

Let the proxy pull values $S_i(t_k), \mu_i(t_k), \sigma_i(t_k)$ from the server at time t_k. $\sigma_i^2(t_k)$ is a reasonable estimate of the current volatility, but will change in the next interval. We use the H-W model to estimate the expected volatility $\hat{\sigma_i}^2(t_{k+1})$ in the next interval. Using finite differences for differentials, we get

$$\hat{\sigma_i}^2(t_{k+1}) = \sigma_i^2(t_k) \cdot (\alpha(t_k)\Delta t_k + 1). \qquad (11)$$

Coefficient $\alpha(t_k)$ can be estimated from the previous obtained volatilities $\sigma_i^2(t_k)$ and $\sigma_i^2(t_{k-1})$ as follows:

$$\alpha(t_k) = \frac{\sigma_i^2(t_k) - \sigma_i^2(t_{k-1})}{\sigma_i^2(t_{k-1})\Delta t_{k-1}}, \qquad (12)$$

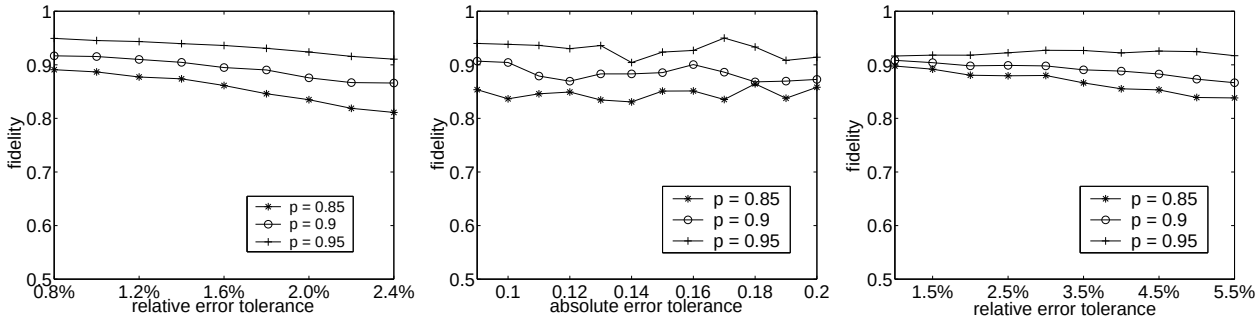
where $\Delta t_{k-1} = t_k - t_{k-1}$. Taking $\Delta t_{k-1} / \Delta t_k \approx 1$, we obtain $\hat{\sigma_i}(t_{k+1}) = \frac{\sigma_i^2(t_k)}{\sigma_i(t_{k-1})}$ from Equation 11 and 12. Now, We can calculate Δt_k as before, using the adjusted diffusion parameter $\hat{\sigma_i}(t_{k+1})$ instead of $\sigma_i(t_k)$.

5.2 Performance of Our Approach

Equipped with the σ adjustment method discussed above, we first demonstrate the *fidelity* of our approach on the three classes of streaming data with both absolute (ϵ) and relative error tolerance ($\epsilon_r\%$). We simulate each data stream as an erratic data object, and only one proxy is simulated in this experiment.

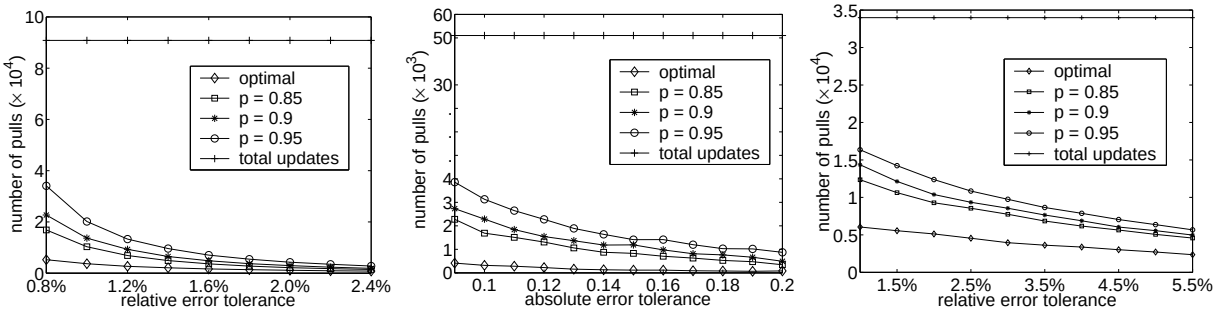
Each point in Figure 4 is the averaged fidelity over five datasets belonging to the same class of data. The fidelity achieved by our model closely matches the prespecified confidence p. For example, the average fidelity at 90% level is 89.4% for stock data ($\epsilon_r\%$), 88.4% for *temp* data (ϵ), and 89.1% for system load data ($\epsilon_r\%$). Such results suggest that our approach captures source data change patterns well, and predicts pull times well. Not surprisingly, a higher p value achieves higher fidelity, but incurs larger communication overhead, as explained below.

Figure 5 demonstrates the communication overhead incurred for maintaining stochastic consistency for different datasets. The communication overhead is measured as the number of pulls generated during the entire simulation period, and varies across the datasets, depending on the dataset's trends and fluctuation patterns. A higher p triggers more pulls, while a looser $\epsilon(\epsilon_r\%)$ triggers fewer pulls. The *total updates* curve shows the total number of data updates in the stream. 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. As can be seen, the communication overhead incurred by our scheme is far less than



(a) stock data(ϵ_r%) (b) *temp* sensor data (ϵ) (c) system load data (ϵ_r%)

Figure 4: Fidelity



(a) stock data: SEBL (ϵ_r%) (b) *temp* data: *0N/140W/36M* (ϵ) (c) system load data (ϵ_r%)

Figure 5: Number of pulls

the naive scheme and close to the optimal scheme.

5.3 Comparison with Adaptive TTL Scheme

A linear adaptive pull scheme (*Adaptive TTL*) with user-provided temporal coherency requirement was introduced in [16, 30]. 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}) [16].

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

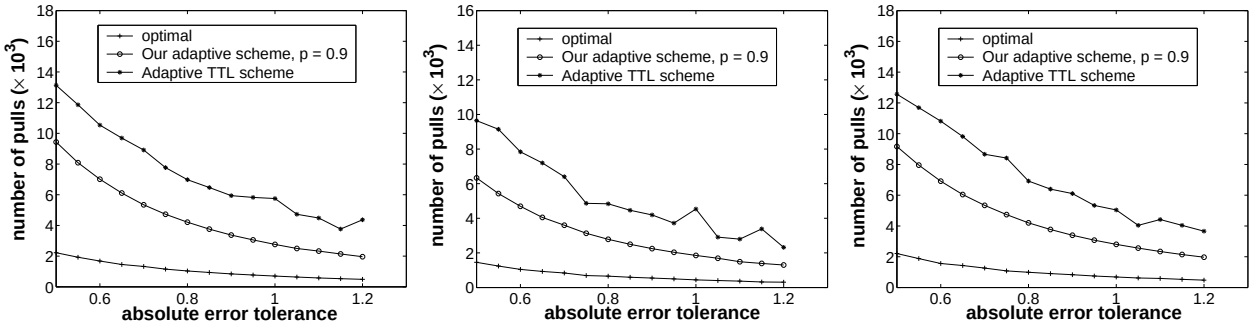
where $T_{dyn} = w \cdot T_{est} + (1-w) \cdot T_l$, $T_{est} = (T_l / |D_{latest} - D_{penultimate}|) \cdot \epsilon$, and $[T_{min},\ T_{max}]$ denotes a static bound for the next TTL value. In [30], the authors compare *Adaptive TTL* with other pull schemes by experiments, and conclude that *Adaptive TTL* outperforms all other schemes.

We compare our scheme with *Adaptive TTL* on various stock traces, in Figure 6. 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.

6 Pull Request Scheduling at Server

Server scalability is one of our major goals, since we want to maximize the number of proxies that a server can handle. If σ_i is high, the request rate for o_i at the server will 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. Thus, we need a scheduler which can intelligently schedule pull requests at the server, so that it guarantees prompt responses whenever possible, and minimizes the total cache-source errors across the proxies when the server is overloaded.

Paradoxically, it is as bad for the server to respond early as it is to respond late. To see this, consider a wireless client which has scheduled the next response from the server to arrive at time t_d. To save power, the client will turn off its wireless receiver, turning it on just before t_d. At time t_d, the client expects the server's response to be "fresh", that is, to contain $S(t_d), \mu(t_d)$, and $\sigma(t_d)$. If the server responds at time



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

Figure 6: Comparing our approach with *Adaptive TTL* scheme on stock traces.

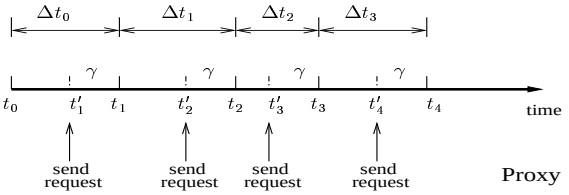


Figure 7: Our scheduling model

$t_a < t_d$, two problems arise. First, the receiver will still be off at t_a, so that this response will be missed entirely. Second, even if we assume that the response can be delivered to the client after it wakes up at t_d, the values $S(t_a), \mu(t_a)$, and $\sigma(t_a)$ will be stale, since the client requires $S(t_d), \mu(t_d)$, and $\sigma(t_d)$. The values $S(t_a)$ and $S(t_d)$ will differ by an amount determined by the DBM model. Early responses by the server cause stale values of S_i, μ_i and σ_i to be delivered at the due times, while late processing will cause cache-source errors to exceed the user-specified bound.

In this section, we introduce a novel variant of *Just-In-Time (JIT)* scheduling, in which a request must complete before its due time, *but as close to it as possible* [21]. We propose a penalty function in Section 6.2, and aim to minimize the overall penalties at any time at the server.

6.1 Scheduling Model Options

Figure 7 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 the $(k+1)$-th request to the 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, we have observed, the server must 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 also track of the Δt_k for a potentially large number of proxies, and can no longer be called stateless, which violates the spirit of the pull model.

Our strategy represents a good compromise. Proxies issue pull requests γ time units before they are really needed, so that the the $(k+1)$-th pull request is made at time $t_{k+1} - \gamma$ (see Figure 7). 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 γ in section 6.4.

Request processing at the server may involve retrieval of the current object value, 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 network models will be our future work.

6.2 Real-Time Scheduling Model

Each pull request, $pull(o_i)$, arriving at the server can be treated a real-time task, parametrized as $T_j(o_i, p_j, d_j)$, where p_j is T_j's processing time; d_j is its due time, $d_j = t + \gamma - \delta$, ($t$ is server's current time, and δ is the proxy-server round-trip network delay. Since the request is sent at time $t_{k+1} - \gamma$, only $\gamma - \delta$ time units are available for server processing.)

As explained before, we must start processing 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 and μ_i, σ_i. Such scheduling is called *Just-In-Time (JIT)* scheduling [21, 24, 25], and penalizes *earliness*, as well as *tardiness*. Tasks finish exactly on their due dates is an ideal *JIT* schedule.

We must first define a penalty function which properly penalizes both tardiness and earliness. If the server's response is tardy, arriving after t_{k+1}, the proxy must continue to estimate the object's value beyond t_{k+1} using the old drift parameter value $\mu_i(t_k)$, although the actual drift and diffusion parameters will have changed to $\mu_i(t_{k+1})$ and $\sigma_i(t_{k+1})$ by t_{k+1}.

Beyond time t_{k+1}, the actual data values change as $S_i(t_{k+1} + \Delta t) = S_i(t_{k+1}) + \mu_i(t_{k+1})\Delta t + \sigma_i(t_{k+1})\Delta W(t)$. However, the proxy has to continue to use $S_i(t_k)$ and $\mu_i(t_k)$ to first estimate $E[S_i(t_{k+1})]$ and then the additional change in this value during the interval Δt as $\mu_i(t_k)\Delta t$. Since t_{k+1} was chosen to make $|S_i(t_{k+1}) - E[S_i(t_{k+1})]| \leq \epsilon$, the extra cache-source value deviation will consist of two parts: a drift-induced component, $|\mu_i(t_{k+1}) - \mu_i(t_k)|\Delta t$, and a diffusion-induced component, $\sigma_i \Delta W(t)$. Since $\Delta W(t)$ is $N(0, \Delta t)$, changes in it can be expressed in terms of its standard deviation $\sqrt{\Delta t}$. Thus, if the due time and the completion time for task T_j are d_j and C_j respectively, the extra error is $|\mu_i(t_{k+1}) - \mu_i(t_k)|(C_j - d_j) + \sigma_i(t_{k+1})\sqrt{C_j - d_j}$.

On the other hand, if the server responds early, at $C_j < d_j$, the proxy receives the values $S_i(C_j), \mu_i(C_j), \sigma_i(C_j)$ at d_j. Consequently, these values are out of date by $d_j - C_j$ time units, resulting in error $\mu_i(C_j) \cdot (d_j - C_j) + \sigma_i(C_j) \cdot \sqrt{d_j - C_j}$.

If the penalty is taken to be proportional to the additional error, The following function penalizes both earliness and tardiness, and accounts for both the drift and diffusion terms. Let μ_j and σ_j are the current drift and diffusion parameters when the pull request T_j arrives the server, and μ'_j is the old drift parameter at the proxy.

$$P_j = \mu_j \max(0, d_j - C_j) + \tilde{\mu_j} \max(0, C_j - d_j) + \sigma_j \sqrt{|C_j - d_j|}, \tag{13}$$

where $\tilde{\mu_j} = |\mu_j - \mu'_j|$. The penalty is zero only when $C_j = d_j$. As C_j deviates more from d_j, more penalty will be incurred. Our goal is to minimize $\sum_j P_j$, the total penalties, at any time at the server. Since the server scheduler needs to know the old drift parameter μ'_j of each pulling proxy, the value of μ'_j should be sent to the server along with each pull request.

JIT scheduling has been well studied, and Baker et al. [21] 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: $h_j \max(0, d_j - C_j) + w_j \max(0, C_j - d_j)$, where h_j is the early cost rate, and w_j is the tardy cost rate. It is known that minimizing an aggregate linear penalty function is NP-hard [25]. Since our version 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 be efficient and on-line, 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 [25].

Theorem 1. *Given n tasks, let the objective be to minimize* $\sum_{j=1}^{n}(h_j \max(0, d_j - C_j) + w_j \max(0, C_j - d_j))$. *Any adjacent pairs of tasks in the optimal sequence must satisfy*

$$w_x p_y - \Omega_{xy}(w_x + h_x) \geq w_y p_x - \Omega_{yx}(w_y + h_y),$$

where task x immediately precedes task y, and Ω_{xy} *and* Ω_{yx} *are defined as:*

$$\Omega_{uv} = \begin{cases} 0 & if\ s_u \leq 0, \\ s_u & if\ 0 < s_u < p_v, \\ p_v & otherwise. \end{cases} \tag{14}$$

Here $s_u = d_u - t - p_u$ *is the slack of task u,* p_u *is its processing time, and t is the current system time.*

Proof: See [25]. □

Theorem 1 offers a necessary condition for an optimal schedule, from which the *LIN-ET* heuristic [25] is derived as follows: given n tasks at time t, they are sequenced in order of priority R_j, calculated as follows:

$$R_j(s_j) = \begin{cases} W_j & if\ s_j \leq 0, \\ W_j - s_j(W_j + H_j)/k\overline{p} & if\ 0 < s_j < k\overline{p}, \\ -H_j & if\ s_j \geq k\overline{p}. \end{cases} \tag{15}$$

where $W_j = w_j/p_j$, $H_j = h_j/p_j$, and $\overline{p}$ is the average processing time of n tasks, and k is called *look-ahead parameter*, used to extend the scope of optimality beyond two adjacent tasks. In practice, as discussed in [25], a low k ($k = 2$, or $k = 3$) may be adequate.

LIN-ET performs quite well under various settings [25], and is computing 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 13) with the priority representation (Equation 15) as follows.

Theorem 2. *Given n tasks, let the objective be to minimize* $\sum_{j=1}^{n}(\mu_j \max(0, d_j - C_j) + \tilde{\mu_j} \max(0, C_j - d_j) + \sigma_j\sqrt{|C_j - d_j|})$. *All adjacent pairs of tasks in the optimal sequence must satisfy*

$$w_x p_y - \Omega_{xy}(w_x + h_x) \geq w_y p_x - \Omega_{yx}(w_y + h_y)$$

where

$$w_x = \tilde{\mu_x} + \frac{\sigma_x}{\sqrt{|s_x|} + \sqrt{|s_x - p_y|}},$$

$$h_x = \mu_x + \frac{\sigma_x}{\sqrt{|s_x|} + \sqrt{|s_x - p_y|}}. \tag{16}$$

The other notations are as in Theorem 1.

We proved Theorem 2 in [35]. Now our scheduling problem can be mapped to the linear penalty case (Theorem 1) with weights w_x and h_x defined in Equation 16. Our heuristic *LINSQT-ET* sequences tasks in order of priority R_j defined in Equation 15, using w_j and h_j as in Equation 16.

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.

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 16 */*
$T_j \cdot priority = W_j - s_j(W_j + H_j)/k\overline{p}$;
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*. */*
    $wait(\omega)$;
  end if

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

According to Equation 15, tasks whose slack times exceed $k\overline{p}$ have the lowest priorities ($-H_j$), 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 round-trip delay, a proxy needs to initiate a pull no earlier than $k\overline{p} + p_j + \delta$ time units ahead of its due time. So we can derive $\gamma = k\overline{p} + p_j + \delta$ in Figure 7.

In Algorithm 2, pending tasks are ordered by their priorities in *PRIO-LIST*. Task priority is a function of the slack time (Equation 16), and changes with time. However, updating priority values and reordering *PRIO-LIST* requires $O(n \log n)$ 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 on-line *LINSQT-ET* scheduling algorithm on stock traces, and compare its performance with the FCFS-based scheduling under various workloads. FCFS scheduling is widely used in current systems, such as Apache [2]. We set the constant lookahead parameter $k = 3$. To calculate γ, proxies obtain the value of k, $\overline{p}$, and p_j the first time they contact the server. We assume each proxy caches one data object, the value of ϵ is randomly chosen from range $0.1 - 0.5$. The request processing time for each proxy is randomly drawn from the range 25ms–40ms [9].

Figure 8(a) compares FCFS and *LINSQT-ET* with respect to the fraction of time the server is overloaded, which is calculated as the percentage of total simulation time that at least one task misses its due time. As the figure shows, the server gets more overloaded as the number of proxies increases in both schemes. However, *LINSQT-ET* remains far more scalable than FCFS throughout the wide range considered. Figure 8(b) compares the two schemes with respect to the average penalties during overload. As can be seen, *LINSQT-ET* scheduling incurs significantly less penalty, and consequently less cache-source deviation than FCFS.

Figure 8(c) compares the penalties incurred by the two schemes relative to the optimal, given a certain number of pending tasks to schedule. The optimal schedule is the sequence of tasks with minimum penalties, which is computed through an off-line exhaustive search. Generally, the deviation between *LINSQT-ET* and optimal is quite low, while the deviation between FCFS and optimal becomes much larger as the number of pending tasks increases.

7 Conclusions

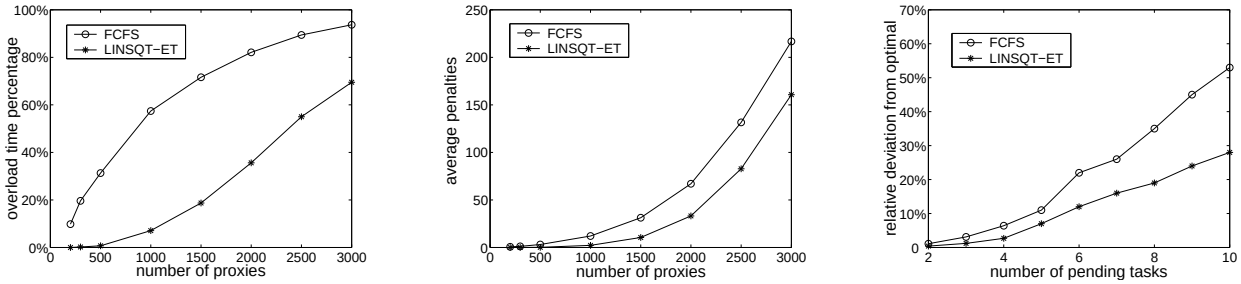
We have proposed *stochastic consistency*, a new model of consistency appropriate for situations when users can tolerate some error. We have also presented a novel pull-based scheme for maintaining stochastic consistency for caches holding erratic and volatile data. Our approach models changes in source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

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 [30], 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.

References

[1] http://mathworld.wolfram.com/erf.html.
[2] http://www.apache.org.
[3] http://www.mesquite.com/htmls/guides.htm.
[4] http://www.pmel.noaa.gov/tao/index.shtml.
[5] http://www.traderbot.com.
[6] Mpr/mib mote sensor hardware users manual. http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_User_Manual_7430-0021-05_A.pdf. Crossbow Inc.



(a) fraction of time server is overloaded (b) average penalties during overload (c) relative deviation from optimal

Figure 8: Performance comparison between FCFS and *LINSQT-ET* heuristic ($p = 0.9$, $\epsilon \in [0.1, 0.5]$)

[7] S. Acharya, M. Franklin, and S. Zdonik. Disseminating updates on broadcast disks. In *Proc. of the 22nd VLDB Conf*, Mumbai, India, September 1996.

[8] S. Acharya, M. Franklin, and S. Zdonik. Balancing push and pull for data broadcast. In *Proc. of the 1997 ACM-SIGMOD Conf*, Tucson, May 1997.

[9] J. Almeida, V. Almeida, and D. Yates. Measuring the behavior of a world-wide web server. Technical Report 1996-025, 29, 1996.

[10] R. Alonso, D. Barbara, and H. G. Molina. Data caching issues in an information retrieval system. In *ACM Trans. Database Systems*, page 15(3). 1990.

[11] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21th ACM SIGACT-SIGMOD-SIGART Symp. on Principle of Database Systems*, Madison, May 2002.

[12] J. Y. Campbell, A. W. Lo, and A. C. MacKinlay. *The Econometrics of Financial Markets*. Princeton University Press, 1997.

[13] J. Cho and H. C. Molina. Synchronizing a database to improve freshness. In *Proc. of the 2000 ACM-SIGMOD conference*, Dallas, May 2000.

[14] J. Cho and H. G. 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.

[15] R. B. D'Agostino and M. A. Atephens. *Goodness-of-fit Techniques*. Marcel Dekker, Inc., 1986.

[16] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic web data. In *The 10th WWW Conference*, Hong Kong, May 2001.

[17] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive leases: A strong consistency mechanism for the world wide web. In *The 19th INFOCOM Conf.*, Tel Aviv, Israel, March 2000.

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

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

[20] R. A. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, 1992.

[21] K.R.Baker and G.D.Scudder. Sequencing with earliness and tardiness penalties: A review. In *Operations Research*, page 38(1). 1990.

[22] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4), 1992.

[23] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the 18th ICDE Conf*, San Jose, 2002.

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

[25] P.S.Ow and T.E.Morton. The single machine early/tardy problem. In *Management Science*, page 35(2). 1989.

[26] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *SIGMOD Conference*, 1991.

[27] S. Saroiu, K. P. Gummadi, R. Dunn, S. D. Gribble, and H. M. Levy. An analysis of internet content delivery systems. In *Proc. of the 5th OSDI Conf*, Boston, MA, December 2002.

[28] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *Proc. of the 28th VLDB Conf*, Hong Kong, 2002.

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

[30] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual data warehouses. In *The 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[31] H. C. Thode. *Testing for Normality*. Marcel Dekker, Inc., 2002.

[32] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proc. of the 14th ICDE Conf*, Orlando, Florida, 1998.

[33] J. Yin, L. Alvisi, M. Dahlin, and A. Lyengar. Engineering server-driven consistency for large scale dynamic web services. In *Proc. of the 10th WWW Conf*, Hong Kong, May 2001.

[34] H. Yu and A. Vahdat. Efficient numerical error bounding for replicated network services. In *The VLDB Journal*, 2000.

[35] S. Zhu and C. Ravishankar. Stochastic consistency, and scalable pull-based caching for erratic data sources. Technical Report UCR-CS-03-85, Univ. of California, Riverside, Nov 2003. http://www.cs.ucr.edu/~szhu/stochpull.pdf.

[36] Y. Zhu and D. Shasha. Statstream: Statistical monitoring of thousands of datastreams in real time. In *Proc. of the 28th VLDB Conf*, Hong Kong, 2002.

[37] H. Zou, N. Soparkar, and F. Jahanian. Probabilistic data consistency for wide-area applications. In *Proc. of the 16th ICDE Conf*, San Diego, February 2000.