Maintaining Mutual Consistency for Cached Web Objects

Bhuvan Urgaonkar, **Anoop George Ninan**, **Mohammad Salimullah Raunak**
Prashant Shenoy and **Krithi Ramamritham**[*]

Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003
{bhuvan,agn,raunak,shenoy,krithi}@cs.umass.edu
http://lass.cs.umass.edu/projects/broadway

Abstract

Existing web proxy caches employ cache consistency mechanisms to ensure that locally cached data is consistent with that at the server. In this paper, we argue that techniques for maintaining consistency of individual objects are not sufficient—a proxy should employ additional mechanisms to ensure that related web objects are mutually consistent with one another. We formally define the notion of mutual consistency and the semantics provided by a mutual consistency mechanism to end-users. We then present techniques for maintaining mutual consistency in the temporal and value domains. Our techniques provide several tunable parameters that allow a tradeoff between network overhead and the fidelity of consistency guarantees. A novel aspect of our techniques is that they can adapt to the variations in the rate of change of the source data, resulting in judicious use of proxy and network resources. We evaluate our approaches using real-world web traces and show that (i) careful tuning can result in substantial savings in the number of polls incurred without any substantial loss in fidelity of the consistency guarantees, and (ii) the incremental cost of providing mutual consistency guarantees over mechanisms to provide individual consistency guarantees is small.

1 Introduction

Web proxy caching is a popular technique for reducing the latency of web requests. By caching frequently accessed objects and serving requests for these objects from the local cache, a web proxy can reduce user response times by up to 50% [12, 16]. However, to fully exploit this benefit, the proxy must ensure that cached data are consistent with that on servers. Several techniques such as *time-to-live (TTL)* values [10], *client polling* [8] and *leases* [6, 18] have been developed to maintain consistency of cached web objects. In this paper, we contend that maintaining consistency of individual objects at a proxy is not sufficient—the proxy must additionally ensure that *cached objects are mutually consistent with one another*. The need for mutual consistency is motivated by the observation that many cached objects are related to one another and the proxy should present a logically consistent view of such objects to end-users. Consider the following examples that illustrate the need for mutual consistency.

- *News articles:* Most newspaper web sites carry breaking news stories that consist of text (HTML) objects accompanied by embedded images and audio/video news clips. Since such stories are updated frequently (as

[*]Also affiliated with the Dept. of Computer Science and Engg., Indian Institute of Technology, Powai, Bombay.

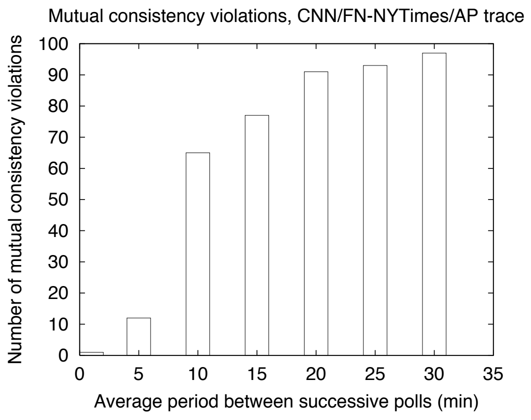


Figure 1: Number of occasions when two cached objects are inconsistent with each other by more than ten minutes. Each violation indicates that the proxy is caching a new version of one object with an older version of the other object for more than 10 minutes.

additional information becomes available), a proxy should ensure that cached versions of such stories and the accompanying embedded objects are consistent with each other.

- *Sports/financial information:* Proxies that disseminate sports information such as up-to-the-minute scores also need to ensure that cached objects are consistent with each other. For instance, a proxy should ensure that scores of individual players and the overall score are mutually consistent. Similarly, a proxy that disseminates financial news should ensure that various stock prices as well as other financial information such as stock market indices are consistent with one another.

To quantitatively demonstrate the need for mutual consistency mechanisms, we considered two time-varying web objects from two newspaper web sites. We assumed that the web proxy employs a cache consistency mechanism to ensure consistency of each individual object[1] but does not employ any additional mechanisms for maintaining mutual consistency. We computed the number of occasions on which the cached versions of the objects were out-of-sync by more than ten minutes with one another. Figure 1 depicts our results. We found that even when each object was polled approximately every 15 minutes, the two objects were inconsistent with one another on about 80 occasions over a 45 hour period. Furthermore, the the number of violations increased as the frequency at which the proxy polled the server was decreased. Since our results indicate that more frequent polls result in fewer mutual consistency violations, one technique to provide mutual consistency guarantees might be to simply poll each object more frequently. However, such an approach can be wasteful in the number of polls, especially if each individual object is changing less frequently. This motivates the need to develop efficient mutual consistency mechanisms that augment existing consistency mechanisms for individual objects.

In this paper, we present adaptive techniques for maintaining mutual consistency among a group of objects. The contributions of our work are three-fold: (i) we identify the need for mutual consistency among web objects, (ii)

[1]The exact mechanism employed to maintain consistency of individual objects is described in Section 3.1.

we formally define the semantics for mutual consistency, and (iii) we propose solutions to provide such consistency guarantees.

We begin by formally defining consistency semantics for individual objects and groups of objects. We argue that mutual consistency semantics are not intended to replace existing cache consistency semantics; rather they augment consistency semantics for individual objects provided by web proxies. Since a mutual consistency mechanism builds upon that for individual consistency, we first propose an adaptive technique for maintaining consistency of individual objects. A novel aspect of our technique is that it deduces the rate at which an object is changing at the server and polls the server at approximately the same frequency (thereby reducing the number of polls required to maintain consistency guarantees). Next, we show how to augment this technique with a mechanism to maintain consistency among a group of objects. Our approach can bound the amount by which related objects are out-of-sync with one another and thereby provide mutual consistency guarantees. Our technique provides tunable parameters that allow network overhead (i.e., number of polls) to be traded off with the fidelity of consistency guarantees.

We demonstrate the efficacy of our approaches through trace-driven simulations. Our simulations are based on real-world traces of time-varying news and financial data and show that: (i) careful tuning can result in substantial savings in the number of polls incurred without any substantial loss in fidelity of the consistency guarantees, and (ii) the incremental cost of providing mutual consistency guarantees over mechanisms to provide individual consistency guarantees is small (even the most stringent mutual consistency requirements result in less than a 20% increase in the number of polls).

The rest of this paper is as structured as follows. Section 2 formally defines the notions of individual and mutual consistency semantics used in this paper. Section 3 presents individual and mutual consistency techniques for the temporal domain, while Section 4 presents these techniques for the value domain. Design considerations that arise when implementing our techniques are discussed in Section 5. Section 6 presents our experimental results. Section 7 presents related work, and finally, Section 8 presents concluding remarks.

2 Individual and Mutual Consistency: Definitions and Approaches

In this section, we first define consistency semantics for individual web objects and then formally define the notion of mutual consistency for web objects.

Consider a proxy that services requests for web objects. To improve the latency for such requests, assume that the proxy caches frequently accessed objects. Cache hits are then serviced using locally cached data, while cache misses require the proxy to fetch the requested object from the server. Typically, the proxy employs a cache consistency mechanism to ensure that users do not receive stale data from the cache. To formally define consistency semantics provided by such a mechanism, let S_t^a and P_t^a denote the version of the object a at the server and proxy, respectively, at time t. The version number is set to zero when the object is created at the server and is incremented on each subsequent update. The version number at the proxy is simply that of the corresponding version at the server. We implicitly require all cache consistency mechanisms to ensure that P_t^a montonotically increases over time. That is, updates from a server should never arrive out-of-order at a proxy and a proxy should never replace a cached object with an older version.

In such a scenario, a cached object is said to be *strongly consistent* with that at the server if the version at the proxy is always up-to-date with the server. That is,

$$\forall t, \quad S_t^a = P_t^a \tag{1}$$

This definition of strong consistency ignores network delays incurred in propagating updates to the proxy. Network delays can be accounted for by modifying the above definition as $\forall t, \quad S_{t-d}^a = P_t^a$, where d denotes the network delay. Strong consistency is difficult to achieve on the Internet due to two reasons. First, due to the large and unbounded message delays on the Internet, a proxy can never strictly guarantee that its version is consistent with the server. Second, network partitions can isolate the proxy from the server and violate consistency guarantees. Furthermore, strong consistency requires that *every* update to the object be propagated to the proxy. This is not only expensive but also wasteful if the proxy is not interested in every single update. The advantage though is that strong consistency does not require any additional mechanisms for mutual consistency, since the definition of strong consistency implies that objects will always be consistent with one another.

Since many web applications are tolerant to occasional violations of consistency guarantees, we can relax the notion of strong consistency as follows. A cached object is said to be Δ-*consistent* if it is never out-of-sync by more than Δ with the copy at the server. Unlike strong consistency, Δ-consistency allows an object to be out of date with the copy at the server, so long as the cached object is within a bounded distance (i.e., Δ) of the server at all times. An important implication of Δ-consistency is that it does not require every update to be propagated to the proxy—*only those updates that are essential for maintaining the bound Δ need to be propagated.* Δ-consistency can be enforced in the time domain or the value domain. In the time domain, it requires that the copy at the proxy be within Δ time units of the server version at all times. That is,

$$\forall t, \quad \exists \tau, \quad 0 \leq \tau < \Delta, \quad \text{such that} \quad S_{t-\tau}^a = P_t^a \tag{2}$$

We refer to these semantics as Δ_t-consistency. To define Δ-consistency in the value domain, let S_t^a and P_t^a denote the *value* of the object a at time t. Then Δ_v-consistency requires that the difference in the values between the proxy and the server versions be bound by Δ. That is,

$$\forall t, \quad \mid S_t^a - P_t^a \mid < \Delta \tag{3}$$

Whereas Δ_v-consistency is meaningful only when the cached object has a value (e.g., stock prices, sports scores, weather information), Δ_t-consistency can be applied to any web object. A number of techniques can be used to enforce Δ-consistency at a proxy.[2] Δ_t-consistency, for instance, can be simply implemented by polling the server every Δ time units and refreshing the object if it has changed in the interim. By pulling an update every Δ time units, the proxy can ignore all other updates that occur between successive polls and yet maintain consistency guarantees. A more efficient mechanism requires the proxy to predict future changes based on past history and poll the server accordingly [14]. For Δ_v-consistency, a proxy must refresh the cached object every time its value at the

[2]In this paper, we consider only proxy-based approaches. Server-based approaches for enforcing Δ-consistency are also possible. In such approaches, the server pushes relevant changes to the proxy (e.g., only those updates that are necessary to maintain the Δ-bound are pushed). The study of such server-based approaches is beyond the scope of this paper.

server changes by Δ. To do so, the proxy needs to track both the frequency of changes of an object as well as the magnitude of each change in order to predict the next time the object will change by Δ [14]. Regardless of the exact approach, all proxy-based mechanisms need to adapt to dynamic changes to the data, since most time-varying web data changes in a random fashion.

Having defined consistency semantics for individual objects, let us now examine consistency semantics for multiple objects. For simplicity, we focus only on two objects but all our definitions can be generalized to n objects. To formally define mutual consistency (M*-consistency*), consider two objects a and b that are related to each other. Cached versions of objects a and b at time t, i.e., P_t^a and P_t^b, are defined to be mutually consistent in the time domain (M_t-consistent) if the following condition holds

$$\text{if } \ P_t^a = S_{t_1}^a \text{and } \ P_t^b = S_{t_2}^b \quad \text{then } |t_1 - t_2| \leq \delta \tag{4}$$

where δ is the tolerance on the consistency guarantees. Intuitively, the above condition requires that the two related objects should have originated at the server at times that were not too far apart. For $\delta = 0$, it requires that the objects should have *simultaneously* existed on the server at some point in the past. Note that mutual consistency only requires that objects be consistent with one another and does not specify any bounds on individual objects and their server versions (i.e., although mutually consistent, the objects themselves might be outdated with their server versions). Consequently, M_t-consistency must be combined with Δ_t-consistency to additionally ensure the consistency of each individual object. This clean separation between M-consistency and Δ-consistency allows us to combine any mechanism developed for the former with those for the latter. It also allows us to easily augment weak consistency mechanisms employed by existing proxies with those for mutual consistency.

Mutual consistency in the value domain (M_v-consistency) is defined as follows. Cached versions of objects a and b are said to be mutually consistent in the value domain if some function of their values at the proxy and the server is bound by δ. That is,

$$\forall t, \ \ |f(S_t^a, S_t^b) - f(P_t^a, P_t^b)| < \delta \tag{5}$$

where f is a function that depends on the nature of consistency semantics being provided. For instance, if the user is interested in comparing two stock prices (to see if one outperforms the other by more than δ), then f is defined to be the *difference* in the object values. Like in the temporal domain, the above definition provides a separation between Δ_v-consistency and M_v-consistency—the former ensures that the cached value of an object is consistent with the server version, while the latter ensures that some function of the object values at the proxy and the server are consistent. Table 1 summarizes the taxonomy of consistency semantics discussed in this section.

There are several possible mechanisms to implement each of the above consistency semantics. We use a metric referred to as *fidelity* to determine the efficacy of a particular mechanism. Fidelity is defined to be the degree to which a cache consistency mechanism can provide consistency guarantees to users. For instance, a Δ_t-consistency mechanism that satisfies Equation (2) for 95% of the time is said to have a fidelity of 0.95. Fidelity can be computed in two different ways. It could be computed either based on the number of instances for which consistency guarantees are violated, or based on the total time for which a cached object is out-of-sync with the server. In this paper, we use both measures of fidelity to quantify the effectiveness of our proposed approaches.

Table 1: Taxonomy of Cache Consistency Semantics

Semantics	Domain	Type	Example
Δ_t	temporal	individual	Object *a* is always within 5 time units of its server copy
M_t	temporal	mutual	Objects *a* and *b* are never out-of-sync by more than 5 time units
Δ_v	value	individual	Value of object *a* is within 2.5 of its server copy
M_v	value	mutual	Difference in values of objects *a* and *b* is within 2.5 of their difference at the server

3 Maintaining Consistency in the Temporal Domain

In this section, we present adaptive techniques for maintaining consistency in the temporal domain based on the definitions in Section 2. We first present a technique for maintaining consistency of individual objects and then show how to augment it for maintaining mutual consistency.

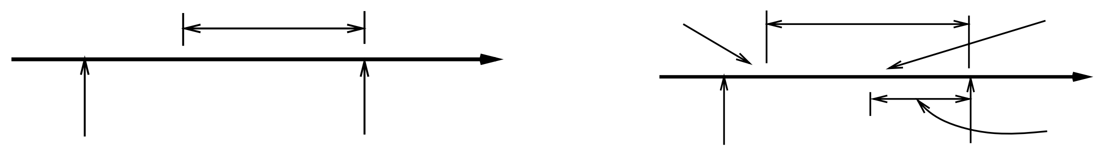
3.1 Maintaining Consistency of Individual Objects

Consider a proxy that caches frequently changing web objects. Assume that the proxy provides Δ_t-consistency guarantees on cached objects. The proxy can ensure that a cached object is never outdated by more than Δ with its server version by simply polling the server every Δ time units (using `if-modified-since` HTTP requests). Whereas this approach is optimal in the number of polls when the object changes at a rate faster than Δ, it is wasteful if the object changes less frequently—in such a scenario, an optimal approach is one that polls exactly once after each change. Consequently, an intelligent proxy can reduce the number of polls by tailoring its polling frequency so that it polls at approximately the same frequency as the rate of change. Moreover, since the rate of change can itself vary over time as hot objects become cold and vice versa, the proxy should be able to adapt its polling frequency in response to these variations.

We have developed an adaptive technique to achieve these goals. Our technique uses past observations to determine the next time at which the proxy should poll the server so as to maintain Δ_t-consistency. To understand the intuition behind our technique, let us refer to the time between two successive polls as the *time to refresh (TTR)* value. [3]

Our technique begins by polling the server using a TTR value of Δ. It then uses a *linear increase multiplicative decrease (LIMD)* algorithm to adapt the TTR value (and thereby, the polling frequency) to the rate of change of the object. If the object remains unchanged between two successive polls, then the TTR value is increased by a linear factor (resulting in less frequent polls). If the proxy detects a violation in the consistency guarantees between successive polls, then the TTR is reduced by a multiplicative factor (causing more frequent polls). If the object is updated between successive polls but does not violate consistency guarantees (indicating the proxy is polling at approximately the correct frequency), then the TTR value is increased gradually until the "correct" TTR is found.

[3]Note that the *Time To Refresh (TTR)* value is different from the *Time to Live (TTL)* value associated with each HTTP request. The former is computed by a proxy to determine the next time it should poll the server based on the consistency requirements; the latter is provided by a web server as an estimate of the next time the data will be modified.



(a) (b)

Figure 2: Possible scenarios that result in violation of consistency guarantees. In scenario (a), the last update occurs more than Δ time units prior to the current poll. In scenario (b), the first update since the last polls occurs more than Δ time units prior to the current poll.

Thus, the basic approach is to react quickly (multiplicative decrease) in the event of a violation and react slowly (linear increase) when there are no violations.

Techniques based on LIMD have been used in many systems to adapt to changing system conditions. To illustrate, TCP employs a congestion control algorithm based on LIMD; the LIMD algorithm is used to compute the congestion window size based on the bandwidth available for the connection [15]. The window size in increased in a linear fashion so long as bandwidth is available and no loss occurs. The window size is reduced by half (a multiplicative decrease) when a loss event occurs (indicating the presence of congestion). Such an algorithm allows TCP to probe the network for the available bandwidth and compute a congestion window based on the current network conditions. Similarly, our algorithm probes the server for the rate at which the object is changing and sets the TTR value accordingly. Due to the adaptive nature of LIMD, the approach can easily handle objects whose rate of change itself varies over time.

The precise LIMD algorithm is as follows. For each object, the algorithm takes as input two parameters: TTR_{min} and TTR_{max}, which represent lower and upper bounds on the TTR values. These bounds ensure that the TTR computed by the LIMD algorithm is neither too large nor too small—values that fall outside these bounds are set to $TTR = \max(TTR_{min}, \min(TTR_{max}, TTR))$. Typically TTR_{min} is specified to be Δ, since this is the minimum interval between polls necessary to maintain consistency guarantees. The algorithm begins by initializing $TTR = TTR_{min} = \Delta$. After each poll, the algorithm computes the next TTR value based on the following four cases.

- **Case 1** *The object did not change since the last poll.* Since the object did not change between successive polls, the TTR is increased by a linear factor.

$$TTR = TTR \cdot (1 + l) \tag{6}$$

  where l is a linear factor and $0 < l < 1$. Consequently, the TTR of a static object increases gradually to TTR_{max}.

- **Case 2:** *The object was modified since the last poll and the consistency guarantees were violated.* Consistency guarantees are violated if the difference between the time of last modification and that of the current poll is larger than Δ (see Figure 2(a)). Violations can occur even when the most recent update is within Δ time units from the poll instant. This is because an object can be modified multiple times between successive polls and the *first* update since the last poll could have occurred more than Δ time units from the current poll instant (see Figure 2(b)). In either case, the TTR is reduced by a multiplicative factor.

$$TTR = TTR \cdot m \tag{7}$$

where m is a multiplicative factor and $0 < m < 1$. Successive violations cause an exponential decrease in the TTR value until it reaches TTR_{min}.

Since each HTTP response is accompanied by the last modification time of the object, a proxy can easily detect violations that fall into the first category (where the most recent update occurred more than Δ time units ago). HTTP responses do not provide any information about updates that occurred *prior* to the most recent change. This makes detection of violations in the second category more difficult. There are two ways to address this problem. First, we can modify HTTP to explicitly provide a history of the most recent changes. We propose such an extension to HTTP in Section 5.1—the enhancement can be made using the user-defined header feature of HTTP version 1.1. Second, the proxy can try to deduce whether a violation occurred. Doing so requires maintaining statistics about past violations so as to infer the probability of a violation.

- **Case 3:** *The object was modified but no violation occurred.* This indicates that the proxy is polling at approximately the correct frequency. The algorithm can then fine-tune the TTR value so as to converge to the "correct" TTR.

$$TTR = TTR \cdot (1 + \epsilon) \tag{8}$$

where ϵ is a small positive number, $\epsilon \geq 0$. By choosing an appropriate value of ϵ, the algorithm can increase the TTR slightly or keep it unchanged.

- **Case 4:** *The object was modified after a long period of no modifications.* This case is handled separately from the previous three cases. If the proxy detects an update after a long period of no modifications, it resets the TTR to TTR_{min}. Since the TTR value is likely to have increased to TTR_{max} in the interim, doing so enables the proxy to handle a scenario where a cold object suddenly becomes popular. If the update was a sporadic event and the object continues to be cold, then the TTR will again gradually increase to TTR_{max}.

Our approach has the following salient features.

- The technique provides several tunable parameters, namely l, m, and ϵ that can be used to control its behavior. In particular, the approach can be made optimistic by employing a large linear growth factor to aggressively increase the TTR value in the absence of updates, and thereby reduce the number of polls. Alternatively, the approach can be made conservative by employing a large multiplicative factor to "back off" quickly in the event of a violation.

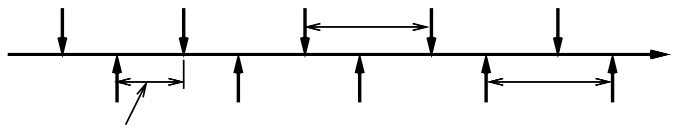


Figure 3: Possible phase lag between polls for two different object.

- An interesting feature of our approach is that it uses information from only the two most recent polls to compute the TTR value. No other information from the past (such as a detailed history of updates or violations) is necessary to compute the TTR. This reduces the amount of state that must be maintained at a proxy and simplifies the proxy design. A further benefit of maintaining minimal state information is that is improves resilience to failures—recovering from a proxy failure simply involves reseting the $TTRs$ of all objects to TTR_{min}.

3.2 Maintaining Consistency Across Objects

Next we present a technique to maintain mutual consistency that augments the Δ-consistency technique presented in the previous section. To understand how such a technique might work, observe that the LIMD algorithm ensures that cached objects are within Δ time units of the server version. In the simple scenario where the algorithm polls the server every $TTR_{min} = \Delta$ time units for each object, two different objects could be out of phase by up to $\Delta/2$ (i.e., the time difference between polls for the two objects could be up to $\Delta/2$ apart). Figure 3 depicts this scenario. The phase lag can be larger when the LIMD algorithm polls each object at a rate slower than Δ. Maintaining mutual consistency requires that polls for related objects should be synchronized ("in-phase") with each other to the extent possible. One approach for doing so is to simply poll each object more frequently—frequent polls reduce the time difference between polls for any two objects and increase the degree of mutual consistency. Another approach is to trigger polls for all related objects every time the proxy polls one of those objects based on the LIMD algorithm; such synchronized polling can ensure that related objects always consistent with one another. The disadvantage of these approaches is that they can significantly increase the number of polls needed to provide consistency guarantees.

Our mutual consistency technique is architected on the observation that polls for related objects need to be synchronized only when one of the objects is updated. In the absence of updates, no mutual consistency guarantees are violated (even if the objects are polled out of phase with one another). Consequently, rather than polling more frequently or synchronizing all polls, the proxy simply employs the LIMD algorithm to maintaining consistency of individual objects. Upon detecting an update (as indicated by the last-modified time field of the HTTP response), the proxy triggers polls for all other related objects. In particular, an additional poll is triggered for an object *only if its next/previous poll instant is more than δ time units away*; no poll is required if the next/previous poll occurs within δ time units, since this is within the user specified tolerance (see Equation (4)).

This approach works well when all objects within a group of related objects change at approximately the same rate. In the scenario where different objects change at different rates, it has the effect of polling all objects at the rate of the fastest changing object. While this approach provides a fidelity of 100% for mutual consistency, it increases the number of polls required to provide mutual consistency guarantees. If the user can tolerate an occasional violation of mutual consistency guarantees (in addition to occasional violations of Δ-consistency guarantees), then a heuristic would be to trigger polls for only those objects that change at a rate faster than the object that was modified. By not polling slower changing objects, this heuristic depends on the LIMD algorithm to detect updates to less frequently modified objects. Observe that the heuristic can result in a violation of consistency guarantees when a less frequently changing object is indeed updated in conjunction with a more frequently changing object and the polls to the two objects are more than δ apart. Section 6 quantifies the impact of this heuristic on the fidelity of mutual consistency guarantees.

4 Maintaining Consistency in the Value Domain

In this section, we present techniques for maintaining consistency guarantees in the value domain. Like temporal domain consistency, we first describe techniques for maintaining Δ_v-consistency and then show how to augment these techniques for maintaining mutual consistency in the value domain.

4.1 Maintaining Consistency for Individual Objects

Recently, we proposed a technique for maintaining Δ-consistency in the value domain [14]. Our technique, referred to as *adaptive TTR computation*, ensures that the difference in the value of an object at the server and a proxy is bound by Δ. That is, $\forall t,\ |S_t^a - P_t^a| < \Delta$. Our technique provides these guarantees by polling the server every time the value of the object changes by Δ. This is achieved by computing the rate at which the object value changed in the recent past and extrapolating from this rate to determine how long it would take for the value to change by Δ. Thus, the TTR is estimated as

$$TTR = \Delta / r \tag{9}$$

where r denotes the rate of change of the object value and is computed as $r = \frac{|P^a_{current} - P^a_{prev}|}{t_{current} - t_{prev}}$; t_{curr} and t_{prev} denote the times of the two most recent polls and P_t^a denotes the object values obtained from those polls. Figure 4 illustrates this process. The TTR estimate in Equation (9) can be improved by accounting for changes that occurred prior to the immediate past; this is achieved using an exponential smoothing function to refine the TTR value. That is, $TTR = w \cdot TTR + (1-w) \cdot TTR_{prev}$, where w determines the weight accorded to the current and past TTR estimates. This TTR value is then constrained using static upper and lower bounds and weighed against the smallest observed value of TTR thus far. Thus,

$$TTR = \max(TTR_{min}, \min(TTR_{max}, \alpha \cdot TTR + (1-\alpha) TTR_{observed-min})) \tag{10}$$

where α is a tunable parameter, $0 \le \alpha \le 1$.

Like the LIMD algorithm presented in Section 3.1, this TTR computation technique can adapt to the dynamics of time-varying data (by virtue of computing a new rate of change of value after each poll). Moreover, the technique uses the parameters w and α to control the sensitivity of the algorithm to the dynamics of time-varying data.

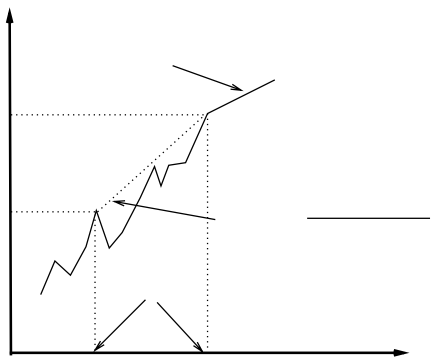


Figure 4: Estimating the rate of change of the object value.

Observe that since a tracked object can change in any random fashion at the server, the efficacy of the above technique hinges on past changes being an accurate indicator of the future. Consequently, greater the temporal locality exhibited by the data, the greater is the effectiveness of the technique. Data that exhibits less locality can be handled by biasing the algorithm towards more conservative TTR values (by picking a small value of α in Equation (10)) and thereby increasing the frequency of polls.

Experiments reported in [14] have demonstrated the efficacy of our adaptive TTR technique in providing consistency guarantees for time-varying financial data (e.g., stock prices). In what follows, we show how to augment this technique to maintain consistency across objects.

4.2 Maintaining Consistency Across Objects

Recall from Section 2 that a mutual consistency mechanism must strive to keep the difference in the values of function f at the proxy and the server within some tolerance δ. That is, $|f(S_t^a, S_t^b) - f(P_t^a, P_t^b)| < \delta$, where f is a function that depends on the values of the two objects. A proxy can achieve this objective by (i) recording recent values of f, (ii) determining the rate of change of the function f and (iii) using this rate to compute the next time instant at which the function f will change by δ. Polling the server before this time instant will ensure that the difference in f at the proxy and the server is bound by δ.

Since the above procedure is similar to that for Δ_v-consistency, we can employ a variant of the adaptive TTR technique described in the previous section to compute poll instants. Thus, given two objects a and b and the tolerance δ, the proxy: (i) polls the corresponding servers for the latest values of a and b, (ii) computes the rate at which the function f is changing as

$$r = \frac{|f(P_{curr}^a, P_{curr}^b) - f(P_{prev}^a, P_{prev}^b)|}{t_{curr} - t_{prev}} \tag{11}$$

and (iii) estimates the TTR value as

$$TTR = \frac{\delta}{r} \cdot c \tag{12}$$

where c is a feedback factor that is varied depending on the accuracy of the TTR estimate, $0 < c \leq 1$. Initially, $c = 1$; c is decreased when consistency guarantees are violated, resulting in more conservative TTR estimates and more frequent polls; c is increased gradually in the absence of violations, causing less frequent polls. Like in the adaptive TTR technique, the TTR estimate in Equation (12) is then refined using Equation (10).

Since we make no assumptions about the nature of the function f, this approach works well only if f is a linear function or if the time difference between successive polls is small enough to approximate f as a linear function. If neither assumption holds, then a different mechanism is required to estimate the rate of change of the the function f.

The efficacy of the TTR estimates can be improved by exploiting the nature of function f. For instance, if f is defined to be the difference in the objects values a and b, then the M_v-consistency condition in Equation (5) reduces to $|(S_t^a - S_t^b) - (P_t^a - P_t^b)| < \delta$, which in turn simplifies to $|(S_t^a - P_t^a) + (P_t^b - S_t^b)| < \delta$. Consequently, a proxy can partition the tolerance δ into two parts δ_a and δ_b, such that $\delta_a + \delta_b = \delta$ and ensure consistency of each individual object using the adaptive TTR approach[4] (i.e., ensure $|P_t^a - S_t^a| < \delta_a$ and $|P_t^b - S_t^b| < \delta_b$ individually).

Parameters δ_a and δ_b can be adjusted periodically based on the dynamics of the objects—a smaller tolerance can be apportioned to the object that is changing at a faster rate, i.e., if r_a and r_b denote the rates of change of objects a and b, then

$$\delta_a = \left(\frac{r_b}{r_a + r_b}\right) \cdot \delta \text{ and } \delta_b = \left(\frac{r_a}{r_a + r_b}\right) \cdot \delta$$

Thus, when f is the difference function, the mutual consistency reduces to maintaining consistency for each individual object, subject to the condition that the summation of individual tolerances is no greater than the total tolerance δ. In cases where the nature of f does not permit any further simplification, then the more general technique described in Equations (11) and (12) must be employed to provide consistency guarantees.

5 Design Considerations

In this section, we discuss design considerations that arise when implementing the cache consistency mechanisms described in this paper. All of our cache consistency mechanisms are based on HTTP. We assume that web users send HTTP requests to their local proxy. Cache hits are serviced using locally cached data, while cache misses cause the proxy to fetch the requested object from the server via HTTP. All of our cache consistency mechanisms compute TTR values for each cached object. The proxy must refresh each object when its TTR expires—this is achieved using an if-modified-since HTTP request, which allows the proxy to query the server if the object is still fresh. Such requests include the time of the previous refresh so as to enable the server to determine if the object has been updated in the interim. In what follows, we discuss some issues that arise when implementing such an approach.

5.1 HTTP Extensions to Provide Update History

The HTTP/1.1 protocol allows a server to provide the time of last modification along with each HTTP request [7]. This information is used by proxies to compute heuristic expiration times (i.e., TTRs) or to test whether a cached object is consistent. The HTTP protocol, however, does not provide any means to include a modification history (in addition to the time of last modification). Such a history, if available, can be used by a proxy for a variety of reasons. For instance, it could be used by proxies to determine frequency of updates to an object and thereby improve the

[4]It follows from elementary algebra that maintaining individual consistency in this manner implies mutual consistency. Since $|x + y| \leq |x| + |y|$, we have $|(S_t^a - P_t^a) + (P_t^b - S_t^b)| \leq |P_t^a - S_t^a| + |P_t^b - S_t^b| < \delta_a + \delta_b = \delta$.

Table 2: User-defined extensions to HTTP/1.1 to provide modification history

entity-header	=	$\cdots$ | Last-modified | extension-header
extension-header	=	Previous-modified-list
Previous-modified-list	=	$*$(Previous-modified)
Previous-modified	=	"Prev-modified" ":" HTTP-date

accuracy of the TTR estimates. In the specific instance of our LIMD approach, such a history could be used to test for violations in Δ-consistency guarantees (see Figure 2(b)). We propose an extension to the HTTP protocol to provide such a modification history using the user-defined header feature of HTTP. Our extension consists of a list of previous-modification times, which is a history of updates in descending order. The list can be of arbitrary length; the server designers may choose a particular length of the history based on practical considerations. The precise syntax for these extensions is specified in Table 2. Observe that HTTP allows proxies and servers that do not understand these user-defined headers to ignore them, thereby enabling backward compatibility with the existing web infrastructure.

5.2 Specifying Tolerances on Consistency Guarantees

Our cache consistency mechanisms require a tolerance Δ to be specified for each cached object and a tolerance δ to be specified for each group of related objects. Depending on the environment, these tolerances could be specified either by a system administrator at proxy configuration time, or by the end-user. Doing so in the former scenario is easy—the administrator simply specifies a single group of tolerances for all objects while configuring the proxy. In the latter scenario, the end-user must somehow communicate his/her preferences to the proxy. This could be done using the cache control directives of HTTP/1.1. These directives allow a user to specify the maximum tolerances on the freshness of the object (by appending tags such as `max-age=`n, `max-fresh=`n and `max-stale=`n with each HTTP request). HTTP currently does not include tags for mutual consistency, but does permit extensions to cache control directives [7]. Consequently, it is easy to add a new directive that allows mutual consistency tolerances to be specified (e.g., using a new tag such as `max-mutual=`n). We envisage a scenario where users will specify these preferences in their browser and the browser will then append appropriate directives with each request. Since different users can specify different tolerances for the same object, the proxy will need to utilize the most stringent of the specified tolerances to actually maintain consistency.

5.3 Determining Groups of Related Objects

Our mutual consistency techniques implicitly assume that relationships among cached objects is known to the proxy. In practice, each set of objects that are related to one another (and desire mutual consistency guarantees) can either be specified by a user or be automatically deduced by the proxy. The latter approach requires a proxy to parse each html object to determine embedded objects and links. This information is then used to construct groups of related objects. While such an approach requires no human intervention, it only captures syntactic relationships between objects

Table 3: Characteristics of Trace Workloads for Temporal Domain Consistency

Trace	Time Period	Num. Updates	Avg. Update Frequency
CNN Financial News Briefs	Aug 7 13:04 - Aug 914:34	113	every 26 min
NY Times Breaking News (AP)	Aug 7 14:07 - Aug 9 11:25	233	every 11.6 min
NY Times Breaking News (Reuters)	Aug 7 14:12 - Aug 9 11:25	133	every 20.3 min
Guardian Breaking News	Aug 6 13:40 - Aug 9 15:32	902	every 4.9 min

and fails to capture semantic relationships (since two objects may be semantically related but may not have links pointing to one another). Manual specification or domain-specific knowledge is the only way to capture semantic relationships. A detailed study of techniques that use syntactic or semantic relationships to construct groups of related objects is beyond the scope of this paper.

6 Experimental Evaluation

In this section, we demonstrate the efficacy of our cache consistency mechanisms through an experimental evaluation. In what follows, we first present our experimental methodology and then our experimental results.

6.1 Experimental Methodology

6.1.1 Simulation Environment

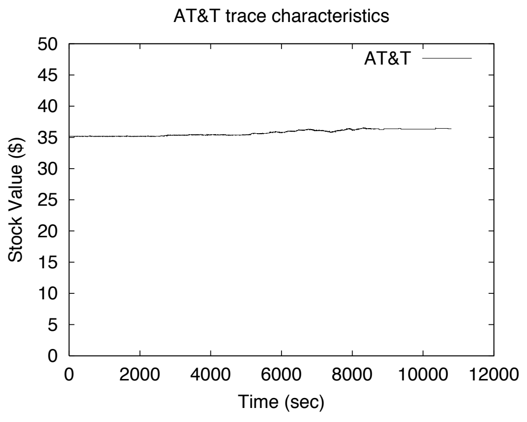
We have designed an event-based simulator to evaluate the efficacy of various cache consistency mechanisms discussed in this paper. The simulator simulates a proxy cache that receives requests from several clients. Cache hits are serviced using locally cached data, whereas a cache miss is simulated by fetching the object from the server.

Our experiments assume that the proxy employs an infinitely large cache to store objects and that the network latency in polling and fetching objects from the server is fixed (this is because we are primarily interested in efficacy of cache consistency mechanisms rather than network dynamics). We assume that tolerances on individual and mutual consistency (i.e., Δ and δ) are specified by the user and known to the proxy.

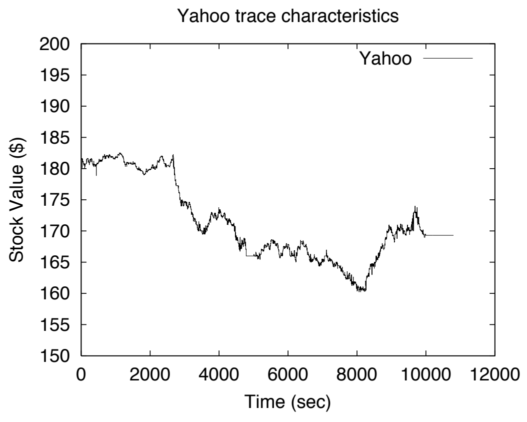
6.1.2 Workload Characteristics

We evaluate the efficacy of our techniques using real-world traces. To evaluate our techniques in the temporal domain, we collected a number of traces from newspaper web sites. To stress our algorithms, we deliberately chose web pages that were updated frequently (i.e., the breaking news sections of these web sites, which are typically updated once every few minutes). We collected our traces using a program that fetched these pages from the server once every minute and determined if the object was updated since the previous poll (by parsing the time-stamp embedded in the html page). The program was run continuously for multiple days on several frequently updated web pages and the characteristics of the resulting traces are summarized in Table 3.

Since value-domain consistency requires web objects that have a value, we chose stock prices to evaluate our techniques. We gathered traces of several stock prices from an online quote server (`quote.yahoo.com`) using



(a) AT&T



(b) Yahoo

Figure 5: Stock price traces for value-domain consistency.

Table 4: Characteristics of Trace Workloads for Value Domain Consistency

Stock Name	Time Period	Num. of Updates	Min Value	Max Value
AT&T	May 22 13:50-16:50	653	\$35.8	\$36.5
Yahoo	Mar 30 13:30-16:30	2204	\$160.2	\$171.2

our trace-collection program. We then choose two particular stock traces for our experiments—one characterized by frequent changes (Yahoo) and the other characterized by infrequent changes in value (AT&T). Figure 5 plots these stock traces and Table 4 summarizes their characteristics.

6.1.3 Metrics

Our cache consistency mechanisms are evaluated using the following metrics: (i) number of polls incurred and (ii) fidelity of the cached data. Fidelity can be measured either based on the number of occasions where consistency guarantees are violated:

$$f = 1 - \frac{\text{Number of violations}}{\text{Number of Polls}} \tag{13}$$

or based on the time for which consistency guarantees were violated:

$$f = 1 - \frac{\text{Total out-sync time}}{\text{Total trace duration}} \tag{14}$$

In general, larger the number of polls, smaller are the chances of violating consistency guarantees. The goal of an effective cache consistency mechanism should be to achieve a fidelity close to 1.0 while incurring as few polls as possible.

6.2 Experimental Results

6.2.1 Maintaining Individual Consistency in the Temporal Domain

We evaluated the LIMD algorithm described in Section 3.1 using our trace workloads. To do so, we configured the algorithm with the following parameters: (i) the linear increase parameter, l, was chosen to be 0.2, (ii) the multiplicative decrease parameter, m, was set to the ratio of Δ and the observed out-sync time , and (iii) the parameter ϵ was chosen to be 0.02. The lower and upper bounds on the refresh interval were set to $TTR_{min} = \Delta$ and $TTR_{max} = 60$ minutes.

We varied Δ from 1 to 60 minutes and computed the number of polls incurred and the fidelity provided by the LIMD algorithm. In each case, we compared our results to a baseline approach where the object was periodically polled every Δ time units. Note that, by definition, this baseline approach always provides perfect fidelity.

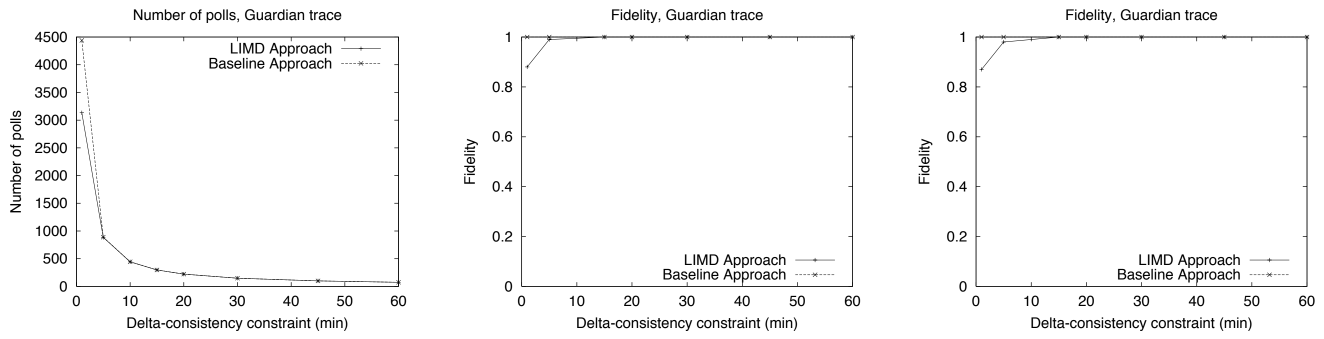
Figures 6 and 7 plot the number of polls and the fidelity for two of our traces, namely the Guardian trace and the CNN/FN trace. We choose these traces since they have the highest and lowest update frequencies; similar results were obtained for other traces, which we omit due to space constraints. These figures indicate the following salient features:

- In the scenario where the consistency requirement Δ is smaller than the interval between successive updates ($\Delta < 4.9$ in Figure 6(a)), the optimal approach is to poll once after each update. The LIMD algorithm attempts to achieve this and consequently incurs a smaller number of polls than the baseline approach (which polls once every Δ time units). However, this reduction in polls comes at the expense of a reduction in fidelity, since the algorithm occasionally fails to detect an update. In the CNN/FN trace, for instance, when Δ = 1 minute, we see a reduction by a factor of 6 in the number of polls with only a 20% loss in fidelity (see Figures 7(a) and (b)). The gains are less dramatic in the case of the Guardian trace (Δ = 1 min yields a 33% reduction in the number of polls with only a 10% loss in fidelity, since fidelity $= 0.9$). Note that, the fidelity can be improved by choosing more conservative values of parameters l and m (which, however, increases the number of polls).

- When the object changes more frequently than Δ, the optimal approach is to poll once every Δ time units. As shown in Figures 6(a) and 7(a), the LIMD algorithm does indeed poll at this frequency (evident from its proximity to the baseline approach). As a result, the fidelity of the LIMD approach converges to that of the baseline approach (namely, a fidelity of 1). Note that the aperiodicity in the CNN/FN trace results in less accurate predictions and more violations, resulting in a slower convergence to the fidelity of the baseline approach. The Guardian trace converges quickly to perfect fidelity due to its periodic, predictable nature.

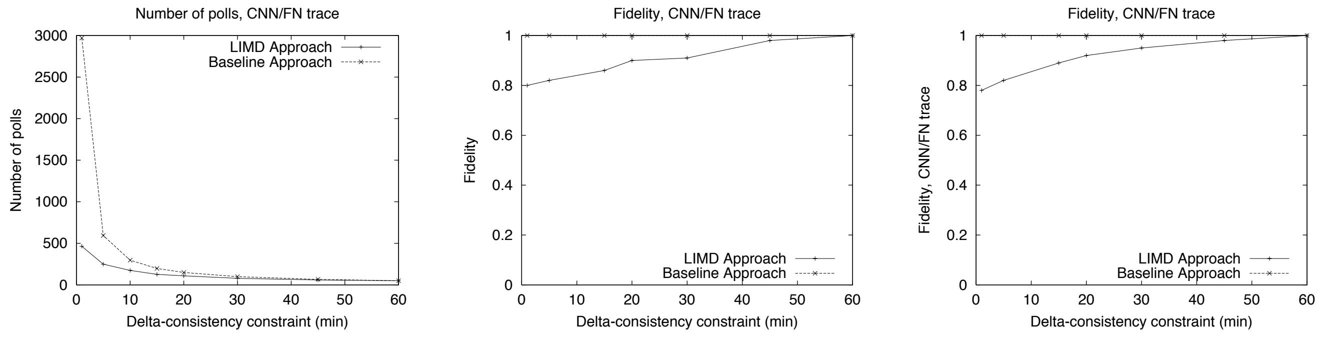
Together, these results demonstrate the adaptive nature of the LIMD algorithm—the algorithm polls less frequently when Δ is less than the update frequency and behaves like the baseline approach when Δ is larger than the update frequency. Note also from Figures 6(b) and (c) as well as Figures 7(b) and (c), that both measures of fidelity demonstrate a similar behavior. Consequently, in the rest of this paper, we only present results for fidelity computed using Equation (13).

The adaptive behavior of our LIMD algorithm is further illustrated in Figure 8. Figure 8(a) plots the average number of updates per 2 hours to the CNN/FN trace. As shown, the update frequency of the CNN/FN web page



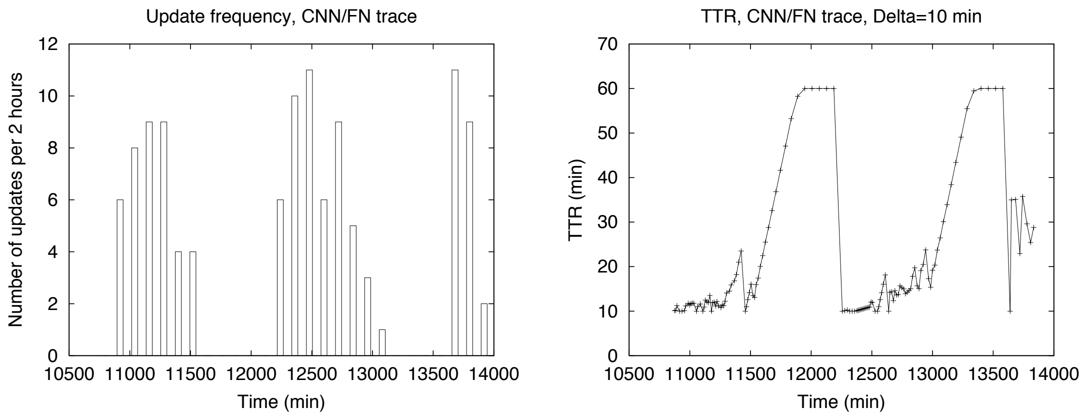
(a) Number of polls (b) Fidelity (num. violations) (c) Fidelity (out-sync time)

Figure 6: Efficacy of the LIMD algorithm for the Guardian trace. The figures compare the number of polls and fidelity offered by the LIMD algorithm to the baseline approach. Note that the baseline approach offers a fidelity of 1.



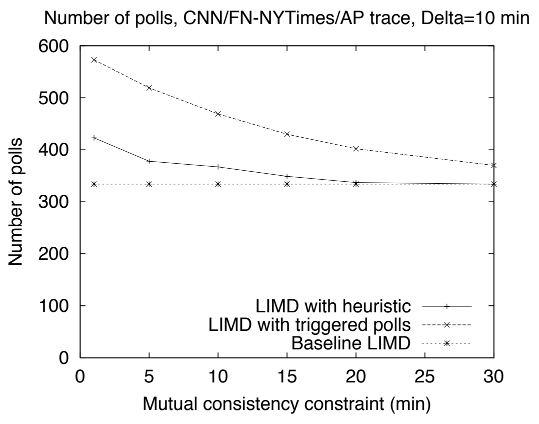
(a) Number of polls (b) Fidelity (num. violations) (c) Fidelity (out-sync time)

Figure 7: Efficacy of the LIMD algorithm for the CNN/FN trace.

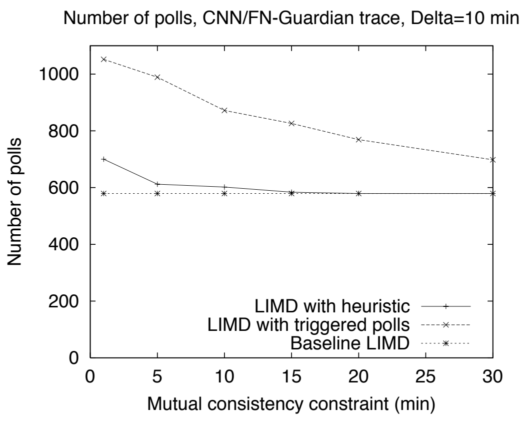


(a) Update Frequency (b) Computed TTR values

Figure 8: Adaptive behavior of the LIMD approach. The TTR increases linearly when the object changes less frequently in the nights. The TTR reduces by a multiplicative factor when updates to the web page are resumed every morning.



(a) CNN/FN and NY Times/AP traces



(b) CNN/FN and Guardian traces

Figure 9: Number of polls incurred for maintaining mutual consistency

reduces to zero for a few hours every night. The LIMD algorithm adapts by increasing the TTR linearly when there are no updates to the object. In particular, the TTR grows linearly to $TTR_{max} = 60$ minutes every night when the object stops changing and reduces in a multiplicative fashion back to $TTR_{min} = \Delta = 10$ minutes every morning.

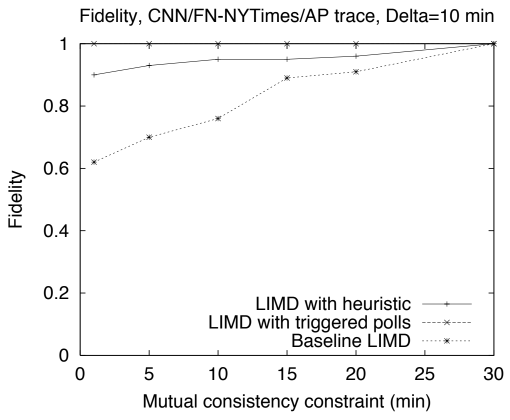
6.2.2 Maintaining Mutual Consistency in the Temporal Domain

Next, we evaluate the efficacy of our mutual consistency techniques. We compare three different approaches: (i) the baseline LIMD algorithm that has no additional support for mutual consistency, (ii) the LIMD algorithm combined with triggered polls, where an update to an object triggers polls to all related objects, and (iii) the LIMD algorithm combined with a heuristic that only triggers polls for objects that change at approximately the same or faster rates.

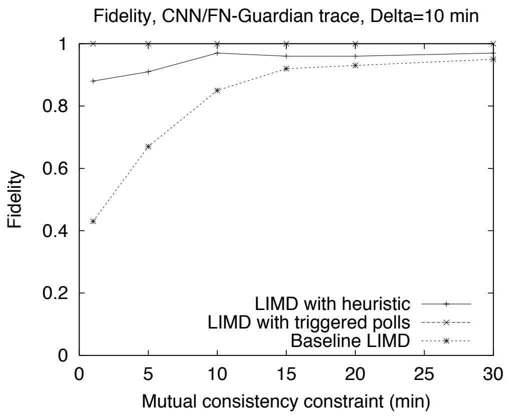
We consider each pair of objects in Table 3 and simulate the above three approaches for mutual consistency (in reality, only the two NY Times traces are related; however, we assume all object pairs are related to enable experimentation with a larger number of object pairs). For each pair of related objects, we varied δ from 1 to 30 minutes and computed the number of polls incurred and fidelity provided by each of the three techniques. In each case, the LIMD algorithm was parameterized by $\Delta = 10$ minutes (other parameters such as l and m were identical to that in Section 6.2.1).

Figure 9 plots the number of polls incurred by the three approaches. As expected, both the triggered poll technique and the heuristic incur more polls than the baseline LIMD algorithm (due to the additional polls required to maintain mutual consistency). Since the heuristic polls selectively based on the update frequency (polls are triggered for only those objects that change at a similar or faster rate), it is more efficient than the triggered poll approach (which polls all related objects regardless of their update frequency). The figure also indicates that the incremental cost of maintaining mutual consistency over maintaining individual consistency is modest. To illustrate, our heuristic results in less than a 20% increase in the number of polls, when compared to the baseline LIMD technique; the overhead is smaller for more tolerant mutual consistency constraints.

Figure 10 plots the fidelity offered by the three approaches. Note that, by definition, the triggered poll technique has a fidelity of 1 (since triggered polls ensure that all related objects are within δ of one another). Depending on

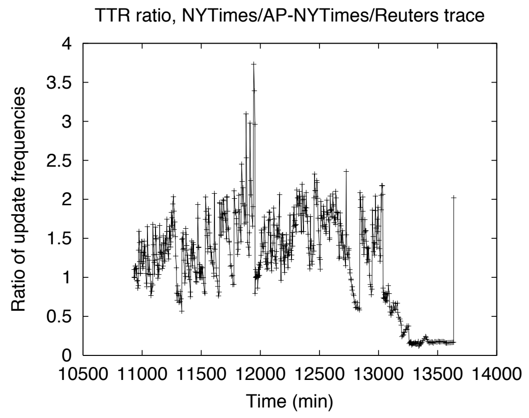


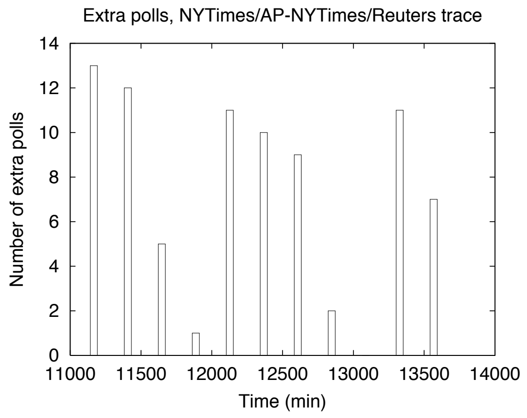
(a) CNN/FN and NY Times/AP traces



(b) CNN/FN and Guardian traces

Figure 10: Fidelity offered by different mutual consistency approaches.



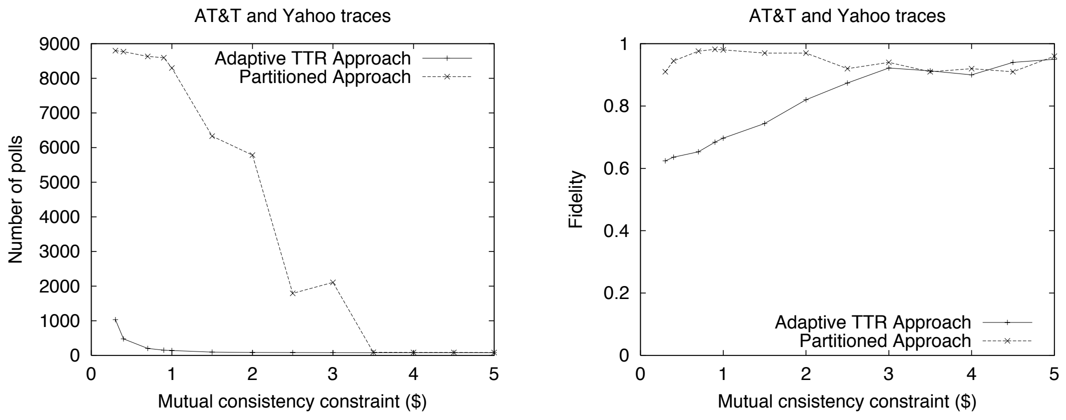


(a) Variation in the ratio of update frequency (b) Number of polls

Figure 11: Adaptive behavior of our heuristic for mutual consistency.

the value of δ, the heuristic approach offers fidelities ranging from 0.87 to 1; higher fidelities are offered for more tolerant values of δ. The figure also indicates that our heuristic is not always successful and can occasionally violate guarantees by triggering polls selectively. Clearly, the baseline LIMD algorithm offers the worst fidelity, since it has no support for mutual consistency. Figure 10 also illustrates that these observations hold when related objects change at approximately the same rates (CNN/FN and NY Times/AP in Figure 10(a)) or at very different rates ((CNN/FN and Guardian in Figure 10(b)).

Finally, Figure 11 illustrates the adaptive nature of our heuristic. The figure shows that an update to one object triggers a poll to the related object only in those time intervals where the related object changes at approximately the same or at a faster rate. For instance, at $t = 11,800$, when the two objects are changing at very different rates (see Figure 11(a)), only the slower object triggers extra polls of the faster object, resulting in fewer polls (see Figure 11(b)).



(a) Number of polls incurred (b) Fidelity

Figure 12: Efficacy of our mutual consistency techniques in the value domain.

6.2.3 Maintaining Mutual Consistency in the Value Domain

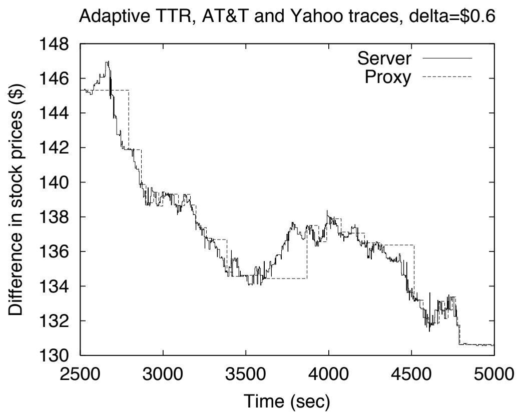
We consider two different approaches to demonstrate the efficacy of our mutual consistency techniques in the value domain: (i) the adaptive TTR approach, which models f as the value of a virtual object and ensures that the value at the proxy is within δ of the server; and (ii) the partitioned approach where we split δ into δ_a and δ_b and reduce the problem to maintaining consistency of individual objects.

We evaluate both techniques using traces of stock prices listed in Table 4 for values of δ ranging from $ 0.25 to $ 5 (mutual consistency requires that the difference in the values of the two objects at the proxy be within δ of their difference at the sever). Figure 12 plots the number of polls incurred and the fidelity offered by two approaches. The figure shows that both approaches incur fewer polls for more tolerant (larger) values of δ; similarly, both approaches offer higher fidelities for more tolerant mutual consistency constraints. Figure 12(b) also shows that by exploiting the nature of the function f, the partitioned approach can offer higher fidelities than the adaptive TTR approach. The approach, however, also incurs a correspondingly larger number of polls to achieve this higher fidelity (see Figure 12(a)). Finally, the fidelities offered by the two approaches is visually illustrated in Figure 13, which plots the values of f at the proxy and the server. As shown, the partitioned approach tracks the server values more effectively when compared to the adaptive TTR approach, resulting in higher fidelities.

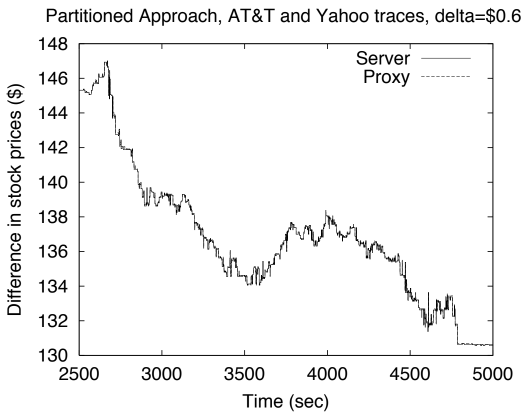
7 Related Work

Whereas cache consistency mechanisms for groups of related web objects have not received much research attention, consistency mechanisms for individual objects have been well studied. Cache consistency mechanisms for individual objects fall into two categories—strong consistency and weak consistency. A consistency mechanism that always returns the result of the latest update at the server is said to be strongly-consistent; mechanisms that do not satisfy this property (i.e., that can occasionally return stale data) are said to be weakly-consistent.

Weak consistency guarantees can be provided by (i) specifying a lifetime for each object (referred to as the *time-to-live (TTL)* value), or (ii) periodically *polling* the server to verify that the cached data is not stale [2, 4, 8,



(a) Adaptive TTR approach



(b) Partitioned approach

Figure 13: Variation in f at the proxy and the server.

13]. In either case, modifications to the object before its TTL expires or between two successive polls causes the proxy to return stale data. Strong consistency can be enforced either by *server-driven* mechanisms or *client-driven* mechanisms [18]. The former approach, referred to as *server-based invalidation*, requires the server to notify proxies when the data changes [3]. The client-driven approach, also referred to as *client polling*, requires that proxies poll the server on *every read* to determine if the data has changed [18]. Whereas server-based invalidation requires state to be maintained at the server, client polling can increase the number of messages exchanged between the server and the proxy and also increases the response time (since the proxy must await the result of its poll before responding to a read request). The leases approach to strong cache consistency balances these tradeoffs by trading state space overhead for control messages and vice versa [18, 6]. In addition to these techniques, cache consistency mechanisms have also been explored in the context of cooperative proxies [1, 17, 19], server volumes [5], delta encoding [11], and cache invalidations [9]. Whereas we have presented our mutual consistency techniques in the context of mechanisms such as our LIMD algorithm, these techniques can also be employed in combination with many of the above approaches.

8 Concluding Remarks

In this paper, we argued that techniques to provide cache consistency guarantees for individual web objects are not adequate—a proxy should additionally employ mechanisms to ensure that related objects are mutually consistent with one another. We formally defined various consistency semantics for individual objects and groups of objects in the temporal and value domains. Based on these semantics, we presented adaptive approaches for providing mutual consistency guarantees in the temporal and value domains. A useful feature of our techniques is that they can be combined with most existing approaches for consistency of individual objects. Our approaches adapt to variations in the dynamics of the source data, resulting in judicious use of proxy and network resources. We evaluated our techniques using real-world traces of time-varying web data. Our results showed that an intelligent proxy could significantly reduce the network overhead in providing mutual consistency guarantees without significantly affecting the fidelity of these guarantees. We also showed that the incremental cost of providing mutual consistency guarantees is small (even the most stringent mutual consistency requirements resulted in less than a 20% increase in the number of polls). As part of future work, we plan to implement our techniques in the Squid proxy cache and demonstrate their utility for real applications.

References

[1] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proceedings of Infocom'99, New York, NY*, March 1999.

[2] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems, Montrey, CA*, December 1997.

[3] P. Cao and C. Liu. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[4] V. Cate. Alex: A Global File System. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

[5] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *Proceedings ACM SIGCOMM'98, Vancouver, BC*, September 1998.

[6] V. Duvvuri, P. Shenoy, and R. Tewari. Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. In *Proceedings of the IEEE Infocom'00, Tel Aviv, Israel*, March 2000.

[7] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Internet-Draft draft-ietf-http-v11-spec-07, HTTP Working Group, August 1996.

[8] J. Gwertzman and M. Seltzer. World-Wide Web Cache Consistency. In *Proceedings of the 1996 USENIX Technical Conference*, January 1996.

[9] B. Krishnamurthy and C. Wills. Proxy Cache Coherency and Replacement—Towards a More Complete Picture. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS)*, June 1999.

[10] J. Mogul. Squeezing More Bits Out of HTTP Caches. *IEEE Network Magazine*, 14(3):6–14, May 2000.

[11] J C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proceedings of ACM SIGCOMM Conference*, 1997.

[12] M S. Raunak, P Shenoy, P Goyal, and K Ramamritham. Implications of Proxy Caching for Provisioning Servers and Networks. In *Proceedings of ACM SIGMETRICS'2000 Conference, Santa Clara, CA*, pages 66–77, June 2000.

[13] *Squid Internet Object Cache Users Guide*. Available on-line at http://squid.nlanr.net, 1997.

[14] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining Temporal Coherency of Virtual Warehouses. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS98), Madrid, Spain*, December 1998.

[15] W R. Stevens. *TCP/IP Illustrated Volume 1*. Addison Wesley, 1994.

[16] R. Tewari, M. Dahlin, H M. Vin, and J. Kay. Beyond Hierarchies: Design Considerations for Distributed Caching on the Internet. Technical Report TR98-04, Department of Computer Sciences, Univ. of Texas at Austin, February 1998.

[17] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical Cache Consistency in a WAN. In *Proceedings of the Usenix Symposium on Internet Technologies (USEITS'99), Boulder, CO*, October 1999.

[18] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Volume Leases for Consistency in Large-Scale Systems. *IEEE Transactions on Knowledge and Data Engineering*, January 1999.

[19] H. Yu, L. Breslau, and S. Shenker. A Scalable Web Cache Consistency Architecture. In *Proceedings of the ACM SIGCOMM'99, Boston, MA*, September 1999.