Best-Effort Cache Synchronization with Source Cooperation*

Chris Olston and Jennifer Widom
Stanford University
{olston, widom}@cs.stanford.edu

Abstract

In environments where exact synchronization between source data objects and cached copies is not achievable due to bandwidth or other resource constraints, *stale* (out-of-date) copies are permitted. It is desirable to minimize the overall *divergence* between source objects and cached copies by selectively refreshing modified objects. We call the online process of selecting which objects to refresh in order to minimize divergence *best-effort synchronization*. In most approaches to best-effort synchronization, the cache coordinates the process and selects objects to refresh. In this paper, we propose a best-effort synchronization scheduling policy that exploits cooperation between data sources and the cache. We also propose an implementation of our policy that incurs low communication overhead even in environments with very large numbers of sources. Our algorithm is adaptive to wide fluctuations in available resources and data update rates. Through experimental simulation over synthetic and real-world data, we demonstrate the effectiveness of our algorithm, and we quantify the significant decrease in divergence achievable with source cooperation.

1 Introduction

Data caching (or *replication*) is a common technique for reducing the latency to access data from remote sources. Ideally, *cached copies* of data objects are kept transactionally consistent with the *source copies* at all times. In practice, transactional consistency is often sacrificed due to the complexity and cost of the required protocols [22]. Furthermore, even propagating all updates in a nontransactional fashion may be infeasible: data collections may be large or frequently updated, and network or computational resources may be limited.

Situations where exact cache consistency is infeasible can be found in many contexts. As one example, consider sensors that continuously monitor environmental conditions such as sound, wind, vibration, etc. Due to recent advancements, it should soon be possible and relatively cheap to deploy large numbers of battery-powered sensors that communicate via wireless links [12, 15, 21]. Since many thousands of sensors may be involved, sensor readings may change frequently, and available bandwidth tends to be low in wireless environments, it is not generally possible to propagate every new sensor measurement to a central cache for monitoring. Similar problems arise in other environments that use wireless or other low-bandwidth links to maintain replica consistency, such as when volatile data is cached on portable devices such as PDA's.

Even in environments that use conventional wired networking, exact cache consistency may still be infeasible due to large quantities of rapidly changing data. For example, in video conferencing applications (*e.g.*, [11]), the viewer screen can be thought of as a cache that maintains copies of video data generated by remote cameras. Since streaming video data can be very large, it often becomes necessary to allow some staleness on parts of the screen. As a final example, consider the problem of indexing the World-Wide Web. Keeping an up-to-date Web index requires maintaining information about the latest version of every document. Currently, Web indexers are unable to maintain anything close to exact consistency due to an astronomical number of data sources and data that is constantly changing.

In environments such as these, where there are not sufficient network or computational resources to keep up with the data as it changes, it is simply not possible to keep the cache synchronized with remote sources. The result is *stale caching*, in which the cache is permitted to store stale, or out-of-date, copies of source data, as illustrated in Figure 1. In stale caching environments, it is desirable to minimize the inconsistency between data in the cache

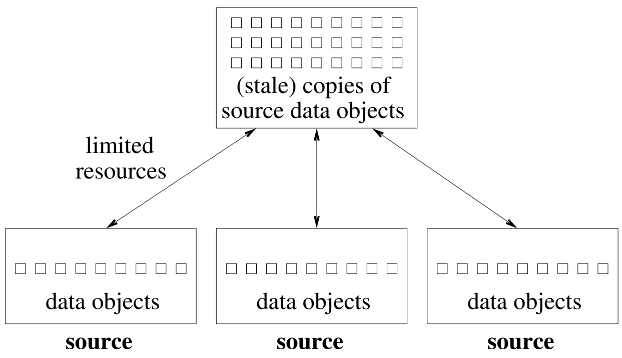

Figure 1: Stale caching architecture.

* This work was supported by the National Science Foundation under grant IIS-9817799 and by a National Science Foundation graduate research fellowship.

ACM SIGMOD 2002 June 4-6, Madison, Wisconsin, USA

and the remote source data. We use the term *best-effort synchronization* for the process of selectively refreshing cached data to maintain the cache as close as possible to exactly synchronized with the sources, in the presence of limited resources.

Note that we use the term *cache* loosely. We assume the cache contains replicas of all source objects of interest (or data derived from source objects, such as an index), and we deal only with the problem of keeping the values of the cached objects up-to-date.

1.1 Source Cooperation

In best-effort synchronization, some policy determines when cached data objects should be *refreshed*. (Remember we are assuming that due to limited resources it is not possible to refresh every object on every update.) In most refresh scheduling policies, *e.g.*, [3, 7], the cache plays the central role: refreshes are scheduled entirely by the cache and implemented by polling the sources, without sources participating in the scheduling. These policies must try to predict which source data objects have changed, and by how much [7, 13]. If source data objects do not behave in predictable ways, the refresh schedule is likely to result in poor synchronization. Since the best synchronization policy obviously depends on how source data objects change, improved synchronization can be achieved through some level of source participation in the refresh scheduling process.

Aside from enabling better synchronization between sources and the cache, there are other, more practical, advantages of source cooperation in synchronization scheduling. First, sources can have a say in weights given to different data objects when prioritizing them for refresh. Moreover, sources can exercise control over the portion of their own bandwidth devoted to cache synchronization, *e.g.*, giving priority to servicing local user queries as they occur and participating in cache synchronization with any spare bandwidth. In contrast, synchronization policies determined entirely by the cache can easily under-utilize available source bandwidth, leading to poor synchronization, or over-utilize source bandwidth, causing a degradation of local processing. This problem is exacerbated when the resources available for synchronization fluctuate over time, *e.g.*, due to sharing network bandwidth, CPU cycles, or disk I/O's with bursty user requests.

1.2 Overview of Approach

In this paper we study the problem of best-effort cache synchronization with source cooperation. We focus on stale caching environments with a large number of sources that synchronize their data with a shared cache. (Recall that we assume the cache contains replicas or derivations of all data objects of interest, *i.e.*, we are not considering cache replacement algorithms.) The resources for cache synchronization may be limited at a number of points. First, the capacity of the link connecting the cache to the rest of the network, the *cache-side* bandwidth, may be constrained. Second, the capacity of the link connecting each source to the rest of the network, the *source-side bandwidth*, may also be constrained and may vary among sources. Moreover, all bandwidth capacities may fluctuate over time if traffic is shared with other applications. We assume a standard underlying network model where any messages for which there is not enough capacity become enqueued for later transmission.

While we cast our approach as coping with limited network resources (bandwidth), our techniques apply more generally to other types of resource limitations. For example, sources may have limited computational resources available for cache synchronization due to local processing load. Caches also may have limited resources for incorporating updates, especially if they perform expensive processing such as data cleaning, aggregation, or index maintenance.

1.2.1 Prioritizing Refreshes

In stale caching, the value of an object at the source and cache may differ. This difference is called *divergence*, and it can be measured using a number of possible metrics including Boolean freshness (up-to-date or not), number of changes since refresh, or value deviation. (We define these metrics formally in Section 3.1.) The best metric to use depends on the data and the caching objectives. Regardless of the divergence metric used, the goal in best-effort synchronization is to minimize the (weighted) sum of the divergence values for each source data object and its cached copy. Weights may be assigned to give certain objects preferential treatment based on criteria such as importance or frequency of access. The choice of divergence metric and weighting scheme should reflect the objectives of the caching environment since those parameters directly affect the synchronization policy. We will revisit these issues in detail later in the paper.

If enough resources are available it is possible to achieve near-zero overall divergence, or even exact transactional consistency if sources will participate in transaction protocols. In environments with limited resources, since not all changes can be propagated, refreshes should be prioritized based on the divergence metric and weighting scheme. Surprisingly, we will see that prioritizing refreshes based solely on the weighted divergence between source and cached copies of data objects does not generally lead to good refresh schedules. We establish a priority policy that achieves much better synchronization. We describe and justify our policy in Sections 3 and 4, respectively.

1.2.2 Coordinating Refreshes Across Multiple Sources

While a good priority policy is an important first step toward best-effort synchronization, it alone is not sufficient. When multiple sources are synchronizing their objects with a shared cache, as in Figure 1, they must share refresh resources such as cache-side bandwidth. Hence, refreshes should be prioritized across all the sources. In the kinds of environments we are considering, sources are not typically aware of the state of the content at other sources. Furthermore, no single entity can keep track of the overall priority order across a large number of sources.

We propose a simple and effective algorithm for scheduling refreshes from a large number of sources that incurs low communication overhead while achieving synchronization that closely follows the global priority order. The idea is for each source to prioritize its own modified objects locally based on the overall priority policy. Ideally, as we will see later, all modified objects having priority above a global *refresh threshold* $\mathcal{T}$ should be refreshed. However, since the best refresh threshold $\mathcal{T}$ varies over time due to fluctuating available bandwidth and divergence rates, measuring the best value for $\mathcal{T}$ and broadcasting it to all sources is impractical, especially when the number of sources to coordinate is very large and bandwidth is limited. Consequently, each source must maintain its own independent copy of the refresh threshold, and some protocol for loosely regulating the individual thresholds needs to be in place.

One way to regulate and coordinate the source refresh thresholds without incurring too much communication overhead is to rely on occasional feedback messages from the cache requesting that sources raise or lower their thresholds. Relying on negative feedback messages from the cache to raise thresholds (in order to reduce the refresh rate) is dangerous since network resources are already overutilized, so unrecoverable network flooding situations can result. Instead, we propose an adaptive threshold-setting algorithm based on positive feedback. In our algorithm, sources by default gradually increase their thresholds, to conservatively reduce the refresh rate in case there is not enough bandwidth. If the cache detects a surplus of bandwidth, it sends positive feedback messages instructing sources to decrease their thresholds thereby increasing the overall refresh rate to fill the surplus.[1] Our general approach is illustrated in Figure 2.

A detailed presentation and justification of our threshold-setting algorithm is given in Section 5. In Section 6, we show experimental evidence that our algorithm achieves low overall divergence without incurring excessive communication overhead, even in environments with a large number of sources and fluctuating resources and data update rates. We also demonstrate quantitatively the advantages of source cooperation in refresh scheduling over having the cache determine the synchronization schedule unilaterally as in [7].

1.2.3 Making Cooperation Appealing

A global priority policy, as we have been assuming, may not be realistic in environments where sources do not agree on the same policy for refresh priority. Moreover, a cache may have criteria for what to maintain up-to-date that conflicts with the objectives of some sources, *e.g.*, when the sources and cache belong to different administrative domains as is common on the Web. In Section 7 we describe how to extend our synchronization techniques to reconcile the potentially different objectives among sources and between sources and the cache.

Since participating in refresh scheduling may be taxing on the computational resources of the sources, in Section 8 we outline lightweight mechanisms for sources to monitor the priorities of modified data objects and schedule refreshes. Techniques for incorporating changes propagated from sources into a cache without disrupting computation at the cache have already been proposed in, *e.g.*, [1, 2].

1.2.4 Bounding Divergence

Finally, in Section 9 we propose a way to provide guaranteed upper bounds on divergence in certain environments. We present a synchronization scheduling policy that minimizes the average upper bound on divergence to suit applications that require strict guarantees about divergence. By contrast, the rest of this paper addresses the related but distinct problem of minimizing the actual divergence, whose value may be unknown to applications accessing cached data.

[1] We differ from the control theory use of feedback terminology, but we feel that "positive feedback" is a good term for increasing the refresh rate.

2 Related Work

A wide variety of work in the literature is related to best-effort cache synchronization to some extent. We outline some of the most relevant work here.

Many stale caching and replication strategies have been proposed. The basic idea is to abandon strict consistency protocols and instead resort to asynchronous propagation of all database updates, *e.g.*, [10, 14, 22], in order to reduce query response time and improve availability. However, all previous approaches we know of do not consider environments in which there is not enough bandwidth to propagate all updates. In limited-bandwidth environments, it sometimes becomes necessary to wait for several updates to an object to accumulate before refreshing, and to explicitly reorder the refreshes to minimize error, as we propose in our approach.

Reference [16] describes strategies for ordering propagations of complex updates from a single source to a cache. However, only the freshness divergence metric is considered, and the focus is not on environments lacking the resources to propagate all updates. Furthermore, [16] does not address the problem of coordinating refreshes from multiple data sources. In the *CU-SeeMe* video conferencing project [11], an application-specific refresh priority scheme is established, but this work also does not address the problem of coordinating refreshes from multiple data sources.

Theoretical algorithms for merging objects from multiple sources in priority order have been proposed in the parallel priority queue research area, *e.g.*, [4, 25]. These algorithms were designed for use in parallel computing environments with high communication throughput, and consequently require tight communication among participants. By contrast, we focus on widely distributed environments with limited communication resources. Also, network flow-control techniques such as TCP/IP have a similar flavor to our refresh coordination algorithm. However, these techniques alone are not sufficient to address our problem because they typically do not address application-level semantics such as an overall priority ranking that is independent of flow rates and queue sizes.

There has been a great deal of work on scheduling events in real-time systems (see [23] for a survey). Most of this work focuses on scheduling events that have strict completion deadlines, and the goal is to minimize the fraction of events that miss their deadlines. By contrast, we consider an environment in which there are no deadlines, and the goal is instead to minimize the time-average of a potentially continuous inconsistency metric.

Finally, several techniques have been proposed to address the problem of minimizing bandwidth utilization and/or query latency in the presence of constraints on the age or accuracy of cached data, *e.g.*, [8, 9, 18, 19, 27, 28]. In this paper we address what is essentially the dual of that problem: maximizing the accuracy of cached data given constraints on available bandwidth.

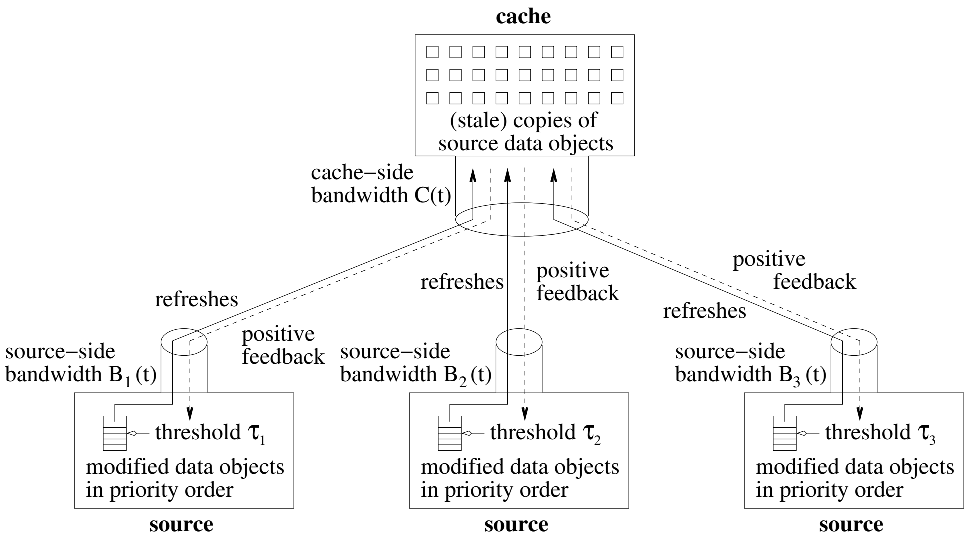

Figure 2: Our approach to best-effort synchronization.

3 Basis for Best-Effort Scheduling

In this section, we begin by formalizing our notion of divergence, then use the formal definition as a basis for a priority policy for best-effort synchronization scheduling.

3.1 Divergence

Consider a source data object O that undergoes updates over time. Let $C(O)$ represent the (possibly stale) cached copy of O. Let $V(O,t)$ represent the value of O at time t. The value of O remains constant between updates. Let $V(C(O),t)$ represent the value of $C(O)$ at time t. Object O can be *refreshed* at time t_r, in which case a message is sent to the cache, and the cached value is set to equal the current source value: $V(C(O),t_r) = V(O,t_r)$. (We assume that the time required to propagate a modified object from a source to the cache is small enough to be neglected.)

In general, let the *divergence* between a source object O and its cached copy $C(O)$ at time t be given by a numerical function $D(O,t)$. When a refresh occurs at time t_r, the divergence value is zero: $D(O,t_r) = 0$. Between refreshes, the divergence value may become greater than zero, and the exact divergence value depends on how the source copy relates to the stale cached copy. There are many different ways to measure divergence that are appropriate in different settings. We define three *divergence metrics* here, but the scope of our work is not limited to these specific metrics.

1. **Staleness**: $D_s(O,t) = 0$ when $V(C(O),t) = V(O,t)$; $D_s(O,t) = 1$ when $V(C(O),t) \neq V(O,t)$.[2]
2. **Lag**: $D_l(O,t) = u$ when $C(O)$ is u updates behind O, *i.e.*, O has been updated u times since the last refresh.
3. **Value Deviation**: $D_v(O,t) = \Delta(V(O,t), V(C(O),t))$, where $\Delta(V_1,V_2)$ can be any nonnegative function quantifying the difference between two versions of an object.

[2]Staleness is the reverse of Freshness ($staleness = 1 - freshness$), which is commonly used in the literature (*e.g.*, [7, 16]). We use staleness so that the larger value corresponds to greater divergence.

When the value deviation metric is appropriate, it usually corresponds to an application-specific function that models some cost associated with the discrepancy between the data value stored at the cache and the actual data value. If the data being cached were Web documents, for example, $\Delta(V_1,V_2)$ might be based on Information Retrieval measures such as TF/IDF vector-space similarity [24]. In the CU-SeeMe video conferencing application [11] mentioned in Section 2, refreshes are prioritized based on the deviation between individual regions of the recorded image and their counterparts on remote viewer screens. The CU-SeeMe value deviation function $\Delta(V_1,V_2)$ is based on the sum of the absolute value of the individual pixel differences, with an additional weight for differences that occur in nearby pixels. In other applications such as stock market monitoring that have single numerical values, the simple value deviation function $\Delta(V_1,V_2) = |V_1 - V_2|$ is often suitable. Once again, note that our techniques are independent of the exact value deviation function or divergence metric used.

3.2 Weights

In many applications, it is desirable to bias the synchronization policy toward refreshing certain important objects more aggressively than others. *Importance* values for objects might be assigned according to various criteria, including but not limited to data quality, content provider authority (*e.g.*, PageRank [3]), and financial considerations. Our approach is independent of the exact importance criteria, but we assume a numerical importance function $\mathcal{I}(O,t)$ that may or may not change over time. In the special case where all objects have equal importance, $\mathcal{I}(O,t) = 1$ for all objects at all times.

In addition to having differing importance, objects also may differ in the frequency with which they are accessed. The *popularity* of an object refers to some measure of the probability of access, possibly weighted by the importance of the person or application that tends to access the data. The popularity of an object O at time t is denoted $\mathcal{P}(O,t)$. In many applications it is important to account for popularity so that scarce resources are used for synchronizing data that will be accessed frequently, maximizing the likelihood of accessing closely synchronized data [16].

From importance and popularity we derive an overall *weight* $W(O,t)$ for refresh assigned to an object O at time t:

$$W(O,t) = \mathcal{I}(O,t) \cdot \mathcal{P}(O,t)$$

There could be other multiplicative factors contributing to $W(O,t)$ besides importance and popularity, based on other aspects relevant to cache synchronization. For example, one could incorporate detailed specifications of the objectives of users as in [5]. For now, we only assume that sources and the cache agree on and are aware of the weighting scheme to be used for best-effort synchronization. In Section 7, we address the possibility of conflicting interests among different sources and between sources and the cache.

3.3 Priority Scheduling

The objective of best-effort synchronization is to minimize the sum of the time-averaged divergence of each object, under the constraint of limited resources [7]. For the staleness divergence metric, this objective is equivalent to minimizing the (possibly weighted) probability of accessing stale data [16]. We begin by studying a theoretical situation in which all sources and the cache share knowledge about each others' state without using network resources, and sources are aware of available cache-side bandwidth. By first considering this idealized situation, we establish an "ideal" scheduling policy for best-effort synchronization, on which we can base our practical techniques.

Assuming for the moment that each source is aware of the state of objects at all other sources, we assert that objects should be prioritized globally for refreshing according to the following formula:

$$P(O_i, t_{now}) = (t_{now} - t_{last(i)}) \cdot D(O_i, t_{now}) \cdot W(O_i, t_{now}) - \int_{t_{last(i)}}^{t_{now}} D(O_i, t) \cdot W(O_i, t)\, dt$$

$P(O_i, t_{now})$ is the *refresh priority* of object O_i at time t_{now}. It is a function of the time $t_{last(i)}$ when O_i was last refreshed, the current time t_{now}, and the divergence and weight of O_i during the interval between $t_{last(i)}$ and t_{now}. The first term is the weighted product of the time interval since the last refresh and the current divergence. The subtracted term is the weighted area under the divergence curve during the interval since the last refresh. The overall priority function $P(O_i, t_{now})$ captures the area above the divergence curve between $t_{last(i)}$ and t_{now}, properly weighted.

The two graphs in Figure 3 depict the refresh priority for two different objects, with time on the x-axis and divergence on the y-axis. Recall that t_{last} denotes the time of last refresh. Object O_1 remained relatively unchanged until recently, then suddenly underwent a significant change. Object O_2 underwent significant changes immediately following the last refresh, but has not changed much since then. In each of the graphs, the area of the shaded region is the unweighted refresh priority for that object. Assuming the two objects are assigned same weight, O_1 will be assigned higher priority for refresh at time t_{now} than O_2.

Intuitively, higher priority is assigned when refreshing an object is likely to have more long-term benefit in terms of divergence reduction. Take object O_1 in Figure 3, which diverged slowly after the last refresh. Assuming it is likely to again diverge slowly if another refresh is performed, a significant reduction in time-averaged divergence can be achieved by refreshing it immediately rather than leaving it with high divergence. On the other hand, object O_2 diverged quickly after the last refresh, so if this behavior repeats itself refreshing O_2 again is likely to have relatively little long-term benefit compared with refreshing O_1, even though they have the same current divergence. Mathematical justification and empirical validation of our refresh priority function are given in Section 4. In [20] we discuss some potential positive and negative implications of extending our priority function to take into account a longer history window.

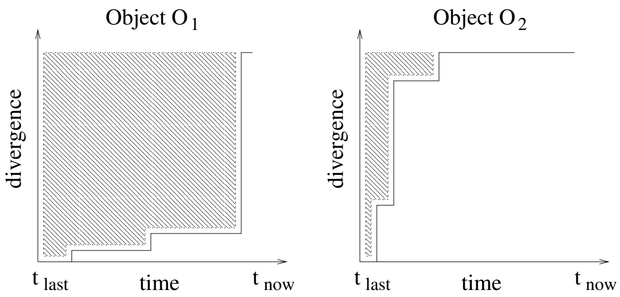

Figure 3: Two divergence graphs showing priority.

Note that in most cases it is reasonable to assume that importance and popularity weights do not change rapidly relative to the time scale at which refreshes occur, *i.e.*, $W(O_i, t) \approx W(O_i, t_{now})$ for all $t_{last(i)} \le t \le t_{now}$. (In fact, in many intuitive weighting schemes, the weights are adjusted very infrequently.) Under this reasonable approximation, we can rewrite the refresh priority function as:

$$P(O_i, t_{now}) \approx W(O_i, t_{now}) \cdot \left((t_{now} - t_{last(i)}) \cdot D(O_i, t_{now}) - \int_{t_{last(i)}}^{t_{now}} D(O_i, t)\, dt \right)$$

Assuming for our idealized scenario that sources know how much cache-side bandwidth is available for refreshes, the ideal synchronization schedule can be achieved as follows. Each time there is enough cache-side bandwidth to accept a refresh, the object with the highest refresh priority among all objects at all sources should be refreshed. If the source containing the highest priority object does not have enough source-side bandwidth available to perform the refresh, then the object with the second highest priority overall should be refreshed instead, and so on.

3.4 Special-Case Priority Functions

The refresh priority formula in Section 3.3 is a general result (justified in Section 4), and applies to any divergence metric. We now give specialized versions of the general priority function for important special cases. Consider a scenario where each object O_i is updated according to a Poisson process with parameter λ_i. In this common scenario (which has been shown to apply to Web pages [7], for example), under the staleness divergence metric specified in Section 3.1, the refresh priority function can be written as:

$$P_s(O_i, t_{now}) = \frac{D_s(O_i, t_{now})}{\lambda_i} \cdot W(O_i, t_{now})$$

The intuition behind this formula is quite simple. First, objects whose cached copies are up-to-date have zero priority, since there is no benefit to repeatedly refreshing the same value. Among objects that are stale, it is desirable to refresh the least frequently changing ones (properly weighted), since they are the most likely to remain up-to-date the longest after being refreshed. In [7], a similar conclusion was reached for the staleness metric in high-contention scenarios. However, our result differs from the exact result presented in [7] because in our scenario, sources have direct knowledge of update times and decide whether to refresh immediately after each update.

Under the lag metric (recall Section 3.1), when updates follow a Poisson model the refresh priority function can be written as:

$$P_l(O_i, t_{now}) = \frac{D_l(O_i, t_{now}) \cdot (D_l(O_i, t_{now}) + 1)}{2\lambda_i} \cdot W(O_i, t_{now})$$

which is roughly proportional to the square of the number of updates to the source value not reflected in the cached copy. This square proportionality indicates that it is especially important to refresh objects that have undergone many changes. Moreover, the priority is inversely proportional to the average change rate λ_i. This inverse proportionality assigns higher priority to objects that are not expected to change rapidly in the future. The derivations of these special-case priority formulae are given in [20].

4 Justification of Refresh Priority Function

In this section we justify, both mathematically and empirically, why prioritizing objects for refreshing using the formulae proposed in Section 3 is appropriate for best-effort synchronization. Let us begin by assuming that bandwidth constraints restrict us to a constant B refreshes/second. Say that there are a total of n objects $O_1, O_2, \cdots, O_n$ among all the data sources. Furthermore, say the divergence of each object O_i depends purely on the time elapsed since the last refresh: $D(O_i, t_{now}) = D^*(O_i, t_{now} - t_{last(i)})$, where $D^*()$ is any nonnegative function. In this scenario, the optimal refresh schedule is one in which each object O_i is refreshed at regular intervals determined by a refresh period T_i.

To determine values for the refresh periods $T_1, T_2, \cdots, T_n$ resulting in the best refresh schedule, we must solve the following optimization problem: minimize the total time-averaged divergence $\overline{D} = \sum_{i=1}^{n}(\frac{1}{T_i} \cdot \int_0^{T_i} D^*(O_i, t)\, dt)$, subject to the bandwidth constraint $\sum_{i=1}^{n} \frac{1}{T_i} = B$. Using the method of Lagrange Multipliers [26], the optimal solution has the property that there is a single constant $\mathcal{T}$ such that for all i:

$$\Phi_i = \mathcal{T} \tag{1}$$

where

$$\Phi_i = T_i \cdot D^*(O_i, T_i) - \int_0^{T_i} D^*(O_i, t)\, dt$$

$\mathcal{T}$ is called the *refresh threshold*, and it controls the overall refresh rate. It corresponds to the (unweighted) priority an object must have in order to be refreshed. A small $\mathcal{T}$ value results in more refreshes, *i.e.*, a high refresh rate. A large $\mathcal{T}$ value results in a low refresh rate. The value of $\mathcal{T}$ depends on the maximum bandwidth B and how fast the objects diverge.

Interestingly, it is possible to discover the optimal refresh policy without directly solving for the refresh periods $T_1, T_2, \cdots, T_n$ if, for all $1 \leq i \leq n$, Φ_i monotonically increases as T_i increases. Under this *monotonicity assumption*, the optimal schedule can be determined online as the current time t_{now} advances by monitoring what the value of T_i would be if object O_i were selected for refresh at the current time: $T_i = t_{now} - t_{last(i)}$. In this scheme, every object O_i would have a proposed refresh period T_i at all times. Given a proposed T_i value for object O_i, Φ_i can be computed using the relationship between t_{now}, $t_{last(i)}$, and T_i along with the relationship between $D()$ and $D^*()$. Note that we are now able to drop the assumption that objects diverge in the same manner after each refresh. We can rewrite Φ_i as the refresh priority at time t_{now}:

$$P(O_i, t_{now}) = (t_{now} - t_{last(i)}) \cdot D(O_i, t_{now}) - \int_{t_{last(i)}}^{t_{now}} D(O_i, t)\, dt \tag{2}$$

Thus, when an object's refresh priority reaches $\mathcal{T}$, that object should be refreshed. Under the monotonicity assumption, the refresh priority of each object monotonically increases with time, so there is exactly one point in time at which the priority equals $\mathcal{T}$, which is the optimal refresh time. By adding weights, we arrive at our original priority function in Section 3.3. In realistic environments, the update patterns of objects and amount of available bandwidth are likely to fluctuate over time, so the best value for the refresh threshold $\mathcal{T}$ changes as well. In Section 5, we give an algorithm for finding and dynamically adjusting $\mathcal{T}$ in a multiple-source environment as bandwidth and update patterns fluctuate.

4.1 Priority Monotonicity

We showed that if priority is expected to increase monotonically, the best time to refresh an object O_i occurs as soon as its priority reaches the refresh threshold $\mathcal{T}$. We now demonstrate that the priority of any object O_i, $P(O_i, t)$, is indeed expected to increase monotonically with time t. Taking the derivative of $P(O_i, t)$ in Equation (2) with respect to time, we obtain:

$$\frac{\partial}{\partial t} P(O_i, t) = (t - t_{last(i)}) \cdot \frac{\partial}{\partial t} D(O_i, t) \tag{3}$$

From this equation, it is easy to see that the expected value of the change in priority $\frac{\partial}{\partial t} P(O_i, t)$ is nonnegative if the expected change in divergence is nonnegative. The latter must be true over time because divergence can never become negative, therefore it must increase at least as much as it decreases. Therefore, unless some special knowledge of future update patterns indicates that an object's source value will converge back toward the cached value, causing divergence to temporarily decrease, priority can be expected to increase monotonically over time.

4.2 Empirical Validation of Priority Function

As discussed in Section 1.2.1, it may appear surprising that it is not a good scheduling strategy to simply prioritize objects according to weighted divergence, *i.e.*, $P(O_i, t) = D(O_i, t) \cdot W(O_i, t)$. To validate our less intuitive priority function empirically, we performed some simulations. We simulated a single data source containing n objects, connected to a cache with bandwidth that supports up to 10 refreshes per second. Each simulated object O_i was updated with probability λ_i each second, and upon each update, the object's value was either incremented or decremented by 1, with equal probability (following a random walk pattern).

In our first experiment, we set all weights to 1 and randomly assigned λ_i values to objects following a uniform distribution. We varied the number of objects from $n = 1$ to 1000 and configured the simulator to prioritize objects for refresh under each of the three divergence metrics: staleness, lag, and value deviation with $\Delta(V_1, V_2) = |V_1 - V_2|$. In all runs, the difference in overall time-averaged divergence observed between our priority function and the simpler alternative was less than 10%.

However, when we introduced some skew into the data parameters, our priority function proved to be significantly better than the simpler alternative. For example, we simulated $n = 100$ objects, a randomly-selected half of which were assigned a weight of 10 while the other half received a weight of 1. An independently- and randomly-selected half of the objects were updated with probability 0.01 while the other half were updated consistently every second. Under the staleness, lag, and deviation metrics, the simple priority function resulted in a 64%, 74%, and 84% increase in overall time-averaged divergence, respectively, compared with our priority function.

5 Threshold-Setting Algorithm

In Sections 3 and 4 we established our approach: prioritize objects and refresh only those whose priority is above a certain refresh threshold $\mathcal{T}$, where $\mathcal{T}$ depends on the available bandwidth and the divergence rates of the objects. Unfortunately, determining the best value for $\mathcal{T}$ would require solving a very large system of equations in most cases: one weighted instance of Equation (1) for each object plus an extra equation for the constraint. Moreover, the available bandwidth and divergence rates may fluctuate widely over time, so most likely there is no single best threshold value that works well all the time. Even if a central site (such as the cache) could gather all the required information and calculate $\mathcal{T}$, if $\mathcal{T}$ changes over time and communication is limited then it may be difficult or impossible to ensure that all m sources are aware of the current threshold value $\mathcal{T}$, especially if the number of sources is very large. In our approach each source S_j maintains its own local refresh threshold value $\mathcal{T}_j$. Whenever a source S_j has enough source-side bandwidth to perform a refresh, it refreshes the object with the highest refresh priority if that priority is above the local refresh threshold $\mathcal{T}_j$.

As the best global threshold $\mathcal{T}$ changes over time, ideally the individual local threshold values $\mathcal{T}_1, \mathcal{T}_2, \cdots, \mathcal{T}_m$ are maintained close to $\mathcal{T}$ to ensure the best overall synchronization schedule. We propose an adaptive algorithm in which the cache and sources work together to adjust the refresh thresholds dynamically, as was illustrated in Figure 2 and discussed briefly in Section 1.2.2. The desired properties of such an algorithm are threefold. First, the algorithm should cause the individual local thresholds to converge on the overall best threshold as conditions change. Second, the algorithm should incur as little communication overhead as possible so as to reserve as much bandwidth as possible for actual refreshes. Third and most importantly, the algorithm must be designed so that it is not possible for a huge excess of refresh messages to become queued in the network for a long period of time. It is crucial to avoid network flooding since refresh messages would be stalled leading to increased cache divergence.

As discussed in Section 1.2.2, the threshold-setting algorithm should avoid relying on negative feedback from the cache. Otherwise, it would be very difficult to recover from situations where the bandwidth is flooded and both refreshes and feedback messages are delayed. A more stable strategy is for the cache to send positive feedback messages when the refresh rate is too slow, asking sources to decrease their thresholds and thereby increase the overall refresh rate. In the absence of feedback, sources can assume that the refresh rate is too fast and should reduce the refresh rate by increasing their thresholds.

In our algorithm, the cache continually monitors cache-side bandwidth utilization. If underutilized, the cache uses the excess bandwidth to send positive feedback messages to as many sources as possible (until the excess bandwidth is utilized), asking them each to decrease their thresholds by a multiplicative factor ω. If it is not possible to provide feedback to every source, the sources with the highest local thresholds are selected to receive feedback. (For the cache to track the source thresholds, each source can piggyback its current local threshold in refresh messages.) When a source S_j receives a feedback message from the cache, it decreases its local threshold $\mathcal{T}_j$ by setting $\mathcal{T}_j := \frac{\mathcal{T}_j}{\omega}$, unless it is already sending at the full capacity of the source-side bandwidth, in which case it leaves $\mathcal{T}_j$ unmodified.[3] In lieu of negative feedback, every time source S_j refreshes an object, it increases its local threshold $\mathcal{T}_j$ by a multiplicative factor $(\theta \cdot \alpha)$ by setting $\mathcal{T}_j := \mathcal{T}_j \cdot (\theta \cdot \alpha)$. Because our algorithm is adaptive, any initial values for the $\mathcal{T}_j$'s can be used and we assume a warm-up period.

The *threshold decrease parameter* ω controls how aggressively the cache requests more refreshes. The *threshold increase parameter* θ controls how quickly sources slow down the refresh rate in the absence of positive feedback. In Section 6.1 we determine good settings for these two parameters. The factor α is used to accelerate the rate of threshold increase in cases where network flooding is likely. If the elapsed time $\Delta t_{feedback}$ since the last feedback message was received at a source is less than the expected feedback period $P_{feedback}$, then $\alpha = 1$. However, whenever $\Delta t_{feedback} > P_{feedback}$, $\alpha = \frac{\Delta t_{feedback}}{P_{feedback}}$. The expected feedback period $P_{feedback}$ is estimated as the ratio of the total number of sources divided by the average cache-side bandwidth. It is not at all critical that the expected feedback period value be exact—it need only be a rough estimate.

6 Experimental Evaluation

We now discuss an experimental evaluation that we performed to determine good settings for the parameters ω and θ, to assess the effectiveness of our algorithm, and to compare against synchronization schedules determined by the cache alone. We constructed a discrete event simulator for an environment with one

[3] We want to avoid situations in which sources have large queues of over-threshold objects due to source-side bandwidth limitations. In such situations, if more source bandwidth suddenly becomes available, sources may flood the cache with refreshes that far exceed the cache bandwidth capacity. If, however, the cache does have plenty of bandwidth available, it will soon send positive feedback messages to the sources, triggering the right amount of additional refreshing.

cache and m sources each containing n objects. In our simulations, the available cache-side and source-side bandwidth fluctuate over time following a sine wave pattern. The average cache-side and source-side bandwidths are controlled by simulation parameters B_C and B_S, respectively. The maximum rate of bandwidth change is controlled by simulation parameter $\Delta_m B$. When $\Delta_m B = 0$, the amount of available bandwidth remains constant. In our simulations, all messages have the same size, and each message requires 1 unit of bandwidth. For most of our experiments, we used synthetic data sets generated following a random walk as described in Section 4.2. Weights vary over time following sine-wave patterns with randomly-assigned amplitudes and periods. We also used one real data set, introduced in Section 6.2.1.

6.1 Parameter Settings

To determine the best settings for the threshold increase parameter θ and decrease parameter ω (Section 5), we performed a variety of simulations. We used synthetic random-walk data generated for a wide variety of configurations having up to $100,000$ objects overall, with fluctuating weights among as many as $m = 1000$ sources. We also varied the amount of cache-side and source-side bandwidth available, where both bandwidth constraints were either held constant ($\Delta_m B = 0$) or allowed to fluctuate over time at a variety of rates. We measured average divergence over a period of 5000 seconds, after an initial warm-up period.

Although our algorithm is not overly sensitive to the parameters θ and ω, it is important to set them carefully. Setting ω too large may cause refresh messages to be sent too aggressively, thereby increasing the latency for refreshes and raising the overall divergence. However, having a small value for ω may lead to underutilization of bandwidth, which also leads to increased divergence. Setting θ too large causes sources to back off on refreshes too quickly, resulting in many positive feedback messages that reduce the bandwidth available for refreshes. On the other hand, setting θ too low sacrifices adaptiveness.

Overall, under all three divergence metrics, we found that the lowest average divergence resulted with threshold increase factor $\theta = 1.1$ and threshold decrease factor $\omega = 10$. With these settings, whenever a source refreshes an object, it increases its local threshold by 10% (or more if $\alpha > 1$ because it detects that the network seems to be flooded). Further, whenever a source receives positive feedback from the cache and it is not sending at maximum source-side capacity, it reduces its local threshold to 10% of its value. The difference in the order of magnitude between θ and ω is due to the fact that increases (due to refreshes) are much more common than decreases (due to feedback). We did not find that our algorithm was overly sensitive to the exact parameter settings (*e.g.*, $\theta = 1.2$ and $\omega = 20$ gave similar results).

6.2 Algorithm Effectiveness

Having determined good settings for the algorithm parameters, we ran a series of simulations comparing the divergence resulting from our algorithm with the divergence resulting from the global policy attainable only in the idealized and unrealistic scenario discussed in Section 3. Our comparison was performed using synthetic random-walk data where each object O_i is randomly assigned a Poisson update rate parameter λ_i. We simulated $m \in \{1, 10, 100, 1000\}$ sources, and varied the number of objects per source: $n \in \{1, 10, 100\}$, giving up to $100,000$ objects total. Objects were assigned weights randomly and weights were allowed to fluctuate over time. The average source-side bandwidth was varied between runs in $B_S \in \{10, 100\}$ and the average cache-side bandwidth was varied in $B_C \in \{10, 100, 1000, 10000, 100000\}$. Finally, the bandwidth change rate was varied between runs in $\Delta_m B \in \{0, 0.005, 0.05, 0.25\}$. We measured the average divergence over a period of 5000 seconds, after an initial warm-up period.

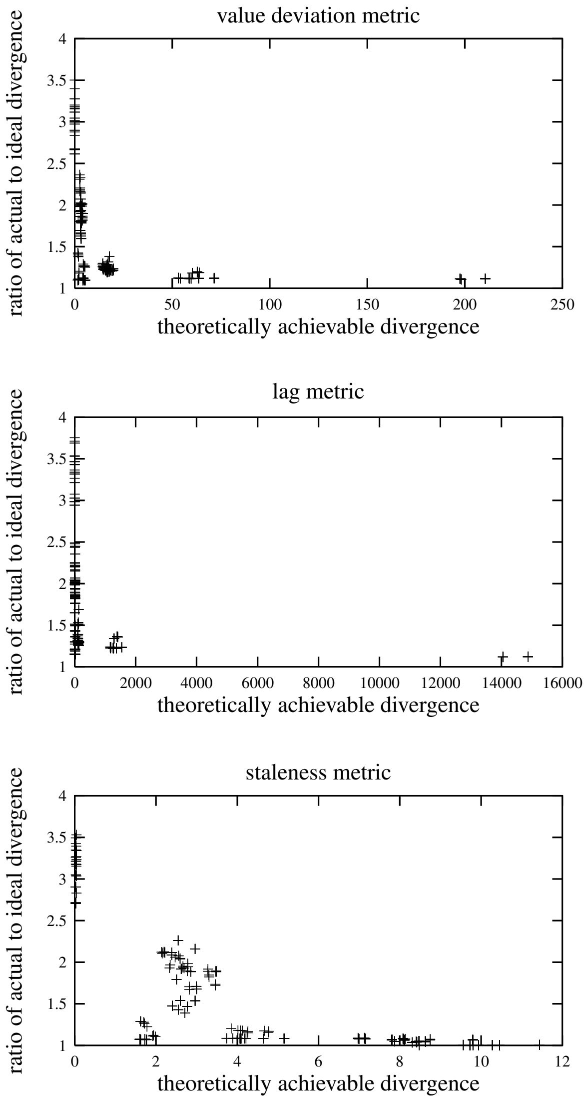

Figure 4: Comparison against the idealized scenario.

Figure 4 shows the results of our experiments using the value deviation, lag, and staleness divergence metrics. One data point is plotted for every combination of the parameters described above. The y-axis shows the ratio of the average divergence resulting from our pragmatic algorithm to the average divergence theoretically attainable in the idealized scenario. Data points are arranged along the x-axis according to the theoretically attainable average divergence. The actual divergence values along the x-axis reflect the weighting scheme and vary depending on the bandwidth avail-

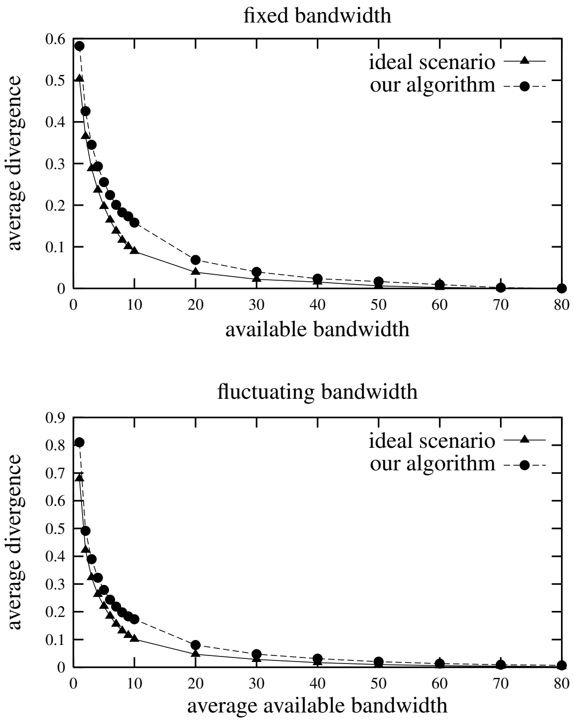

Figure 5: Average divergence over wind buoy data.

ability relative to the data update rates, so they are not particularly meaningful.

From Figure 4, we can see that as the average theoretically attainable divergence increases (due to low bandwidth and/or many rapidly diverging objects), our algorithm attains divergence nearly as good as the ideal case. On the other hand, when divergence is small, the absolute difference between the divergence achieved by our algorithm and that of the idealized case is small. Overall, our algorithm results in divergence that is close to that theoretically attainable in the idealized case. Sections 6.2.1 and 6.3 give further evidence to support this claim.

6.2.1 Effectiveness on Real-World Data

To further verify the effectiveness of our algorithm, we performed some experiments on a real-world data set gathered from weather buoys in January 2000 by the Pacific Marine Environmental Laboratory [17]. We simulated monitoring wind vectors from $m = 40$ buoys spread out in the ocean, which perform measurements every 10 minutes. Each wind vector is made up of two numeric components, giving $n = 2$ data values per data source (buoy). All data values were equally weighted.

Using the value deviation divergence metric with $\Delta(V_1, V_2) = |V_1 - V_2|$, we simulated seven days worth of wind data, using the first day as a warm-up period. The maximum total number of messages transmitted per minute over the satellite link (cache-side bandwidth) was constrained. In the graphs in Figure 5, the (average) maximum bandwidth is plotted on the x-axis and the resulting average value deviation per data value is shown on the y-axis. The first graph shows the results of experiments in which the maximum bandwidth was fixed as a constant between 1 and 80. In the second graph, available bandwidth fluctuated with time following a sine wave pattern with a peak relative change rate of $\Delta_m B = 0.25$. The wind velocity values monitored were generally in the range of 0–10, with typical values of around 5, so 0.5 on the y-axis for example indicates roughly 10% divergence. Figure 5 shows that the divergence achieved by our threshold-setting algorithm closely follows the divergence theoretically achievable in the idealized scenario.

6.3 Comparison Against Cache-Based Scheduling

Finally, to quantify the benefits of source cooperation in synchronization scheduling, we compared our cooperative approach against a recent fully cache-driven approach by Cho and Garcia-Molina [7]. In their approach, which we will refer to as “CGM,” the cache schedules all refreshes and polls sources for values. The refresh frequency for each object O_i is set independently based on an estimate of its average update rate λ_i. The goal is to minimize the staleness metric (without weights) and the overall bandwidth utilization is controlled by a numeric parameter μ, which was shown not to be solvable mathematically [7]. The CGM policy was shown to be the optimal cache-based synchronization scheduling policy, given the correct setting for μ [7]. In our experiments, we used repeated runs to experimentally determine the correct setting for their parameter μ.

Our comparison was performed over synthetic random-walk data where each object O_i is randomly assigned a Poisson update rate parameter λ_i. Since the polling model used in the CGM approach assumes no limitations on source-side bandwidth, we only placed a limitation on cache-side bandwidth, which we varied between runs. We simulated $m \in \{10, 100, 1000\}$ sources, with $n = 10$ objects per source (results for $n = 100$ objects per source were similar). We varied the bandwidth capacity between 10% and 90% of the total number of objects (*i.e.*, between $0.1 \cdot m \cdot n$ and $0.9 \cdot m \cdot n$) between runs. Since the CGM approach assumes a fixed amount of available bandwidth, this quantity was held constant during each run (*i.e.*, $\Delta_m B = 0$). We measured the average unweighted staleness over a period of 500 seconds, after an initial warm-up period. (We used a shorter measurement period in this experiment than in previous ones since the bandwidth doesn’t fluctuate over time.)

Figure 6 shows the results of our comparison for $m = 10$, 100, and 1000 sources. In each graph, the x-axis is bandwidth capacity as a fraction of the total number of objects $m \cdot n$. The y-axis shows average divergence (staleness, in this case), and the five data lines correspond to five different theoretical or practical synchronization techniques. “Ideal cooperative” is the idealized algorithm discussed throughout this paper, “our algorithm” is self-explanatory, and “ideal cache-based” corresponds to CGM under two theoretical assumptions: that the cache can request refreshes without performing any communication to sources, and that the cache is aware of the exact update rates (λ values) of all of the objects. “CGM1” and “CGM2” are practical implementations of the CGM techniques. First, since refreshes require polling, each refresh incurs a round-trip message from the cache to a source. Second, the cache must estimate the object update rates (λ values) based on observations taken during prior refreshes. Two methods for estimating an object’s update rate are suggested in [6]. The first method can be used if the source keeps track of the time at which the most recent update to each object occurred; this ap-

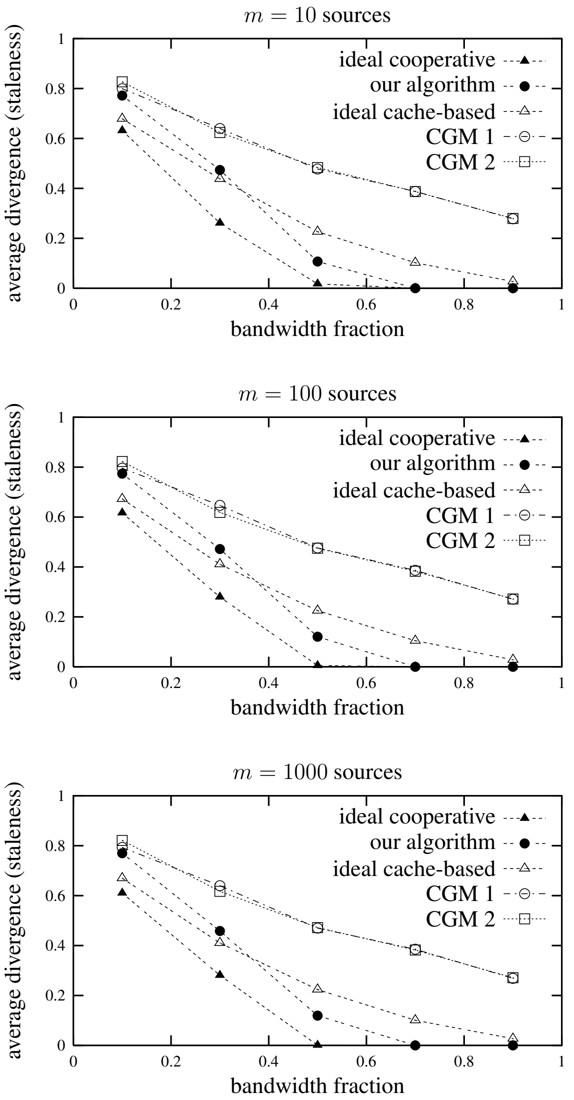

Figure 6: Comparison against cache-based synchronization policies.

proach is CGM1. The second method for estimating update rates is used if the cache can only determine whether an object has been updated since the last refresh, but not when it was updated; this approach is CGM2.

By comparing the "ideal cooperative" and "ideal cache-based" curves in the graphs in Figure 6, we can see that, at least theoretically, cooperative scheduling enables much lower divergence than a cache-based policy. Furthermore, by comparing the curve for our algorithm against the two pragmatic CGM curves, the attainable benefit of cooperative scheduling over cache-based techniques is demonstrated.

7 Cooperation in Competitive Environments

So far we have assumed that there is a single priority function and refresh policy about which all participants (sources and cache) agree. However, in some environments, sources may differ in their criteria for deciding what content to keep up-to-date at a cache. Moreover, a cache's objectives of what to store and maintain up-to-date may not coincide with the goals of the sources. More concretely, the cache may request that sources implement a certain priority policy, determined by a divergence function and weights, but a given source may prefer a different priority policy derived from its own divergence function and weights. The result is that there may be two conflicting refresh priorities for each object.

As an example, consider a Web indexer, whose objective might be to focus resources on maintaining high-importance or high-popularity Web pages up-to-date in the index. Content providers' criteria for prioritizing pages for synchronization may differ from that of the indexer, and each content provider might have different criteria. For example, a retailer might wish to notify the Web indexer whenever a special offer is added to their Web site, for advertising purposes. In general, if the cache and a source disagree on the best refresh priority policy, how can a compromise be made?

Under conflicting priorities, we can partition resources among satisfying source priorities and satisfying the cache priority. Let Ψ represent the fraction of the cache-side bandwidth dedicated to satisfying source priorities, so $(1 - \Psi)$ is the fraction dedicated to cache priority. The parameter Ψ might be set by the cache administrator. In loosely coupled environments, a relatively large Ψ value can serve as an incentive for data sources to affiliate with the cooperative environment because they will be given an opportunity to keep content they value up-to-date at the cache, even if the cache prefers to focus on different content. There are at least three conceivable ways to divide up the Ψ fraction of the cache-side bandwidth dedicated to fulfilling the needs of sources:

1. All sources are given an equal share.
2. Sources are given a share proportional to the number of cached objects from the source.
3. Sources are given a share proportional to the degree to which the source contributes to satisfying the objectives of the cache.

In options (1) and (2), all participating sources or objects are given equal treatment. In option (3), sources are allocated resources for their own purposes only if they bring significant value to the cache by offering objects that the cache wants to maintain highly synchronized. In our Web index example, in option (3) Web content providers with many documents that the index deems to be of high value would be allocated a relatively large amount of synchronization resources to use as they see fit.

To implement options (1) or (2), the cache can monitor the total available cache bandwidth and inform sources with each feedback message how much bandwidth (in terms of number of refreshes per second) they have been allocated. Then, sources can refresh objects based on their own priority scheme at the rate specified by the cache. The remaining cache bandwidth would be dedicated to refreshes following the cache's priority, using the threshold-based algorithm proposed in Section 5. To implement option (3), sources would be permitted to, on average, piggyback $\frac{\Psi}{1-\Psi}$ objects of their own choosing along with every object refreshed based on the cache's priority using the threshold policy.

8 Priority Monitoring Techniques

In this section, we discuss some practical considerations in how sources monitor the refresh priority of their updated objects. Sources need to detect when an object's priority exceeds the refresh threshold and refresh it, assuming sufficient source-side bandwidth. If source-side bandwidth is a limiting factor, sources can maintain a priority queue so that the highest-priority updated object can be located quickly whenever spare bandwidth becomes available. We first discuss what sources need to do to compute the priority of their objects in Section 8.1, and then discuss when sources should measure the priority in Section 8.2.

8.1 How to Measure Priority

If the lag or staleness metrics are employed and objects are updated according to a Poisson process, then an object's priority depends uniquely on update times and not data. One simple way for the source to track priorities is to monitor when updates occur. The number of updates to an object since the last refresh determines its divergence value. The number of updates divided by the time elapsed since the last refresh gives an estimate for the Poisson parameter λ. Alternatively, the parameter λ may be monitored over a longer period of time. From an estimate for λ and the divergence value, the refresh priority can be computed using the formulae given in Section 3.4. If it is impossible or too invasive to track the exact number of updates, one of the techniques proposed in [6] can be used to estimate λ. If the value deviation metric is employed, we need to compare an object's value with the older cached value to measure its divergence, which determines the priority.

8.2 When to Measure Priority

Surprisingly, although the refresh priority depends on time, an object's priority can only change when an update occurs. Equation (3) in Section 4.1 shows the derivative of priority with respect to time. Note that if divergence remains constant, *i.e.*, $\frac{\partial}{\partial t}D(O_i, t) = 0$, then the priority also remains constant. Thus, an object's priority only changes when its divergence changes, which can only occur as a result of updates to the source object.

Therefore, to track the exact priority of an object, sources only need to recompute the priority when an update is made to that object. Since the priority depends on the integral of the divergence values since the last refresh, the source also needs to maintain a running total of the past divergence values weighted by the amount of time the value was active. The data necessary to compute this running total only needs to be modified each time an update occurs. Detecting updates requires the use of triggers or a similar mechanism. If triggers are not supported or are deemed too expensive, object priority can be monitored more loosely using sampling techniques, discussed in [20].

9 Divergence Bounding

Some applications may require guaranteed upper bounds on the divergence of objects accessed at the cache. For example, it may be important to know with certainty that a data value is below a strict threshold or critical value. We can easily guarantee divergence bounds at the cache when the source objects have known maximum divergence rates. Let L_i be an upper bound on the total time required to refresh object O_i.[4] Let R_i be the maximum divergence rate of object O_i. The upper bound on divergence since the last refresh at time $t_{last(i)}$ is $B(O_i, t_{now}) = R_i \cdot ((t_{now} - t_{last(i)}) + L_i)$. In applications requiring divergence bounds, it may be appropriate to perform best-effort synchronization with the goal of minimizing the upper bounds, instead of minimizing actual divergence values. Substituting $B(O_i, t_{now})$ for $D(O_i, t_{now})$ in our priority function of Section 3.3, we obtain the following optimal priority function for minimizing the sum of the time-averaged divergence bounds, assuming the weights do not change drastically between refreshes:

$$P(O_i, t_{now}) = \frac{R_i \cdot (t_{now} - t_{last(i)})^2}{2} \cdot W(O_i, t_{now})$$

The threshold-based algorithm from Section 5 for coordinating refreshes from multiple sources can be used in conjunction with this priority policy.

10 Summary

We proposed, mathematically justified, and empirically verified an algorithm for best-effort cache synchronization with source cooperation. Source cooperation in the synchronization process is advantageous for a number of reasons. First, source cooperation enables better scheduling policies than would otherwise be possible, resulting in improved synchronization over cache-centric approaches. Second, sources can be given a say in the relative priority of their objects for synchronization. Finally, sources can exercise fine-grained control over the source-side bandwidth used for cache synchronization so that exactly the right amount of bandwidth can be devoted to servicing user queries.

We began by defining and justifying a priority policy for refreshing cached objects when bandwidth is limited. We then proposed an algorithm for implementing the policy, while regulating the synchronization rate to match the available bandwidth without excessive communication. Our algorithm adjusts local refresh thresholds adaptively at a large number of data sources as conditions fluctuate. We presented simulation results on both synthetic and real-world data sets to demonstrate that our techniques are effective. We also demonstrated empirically that source cooperation in synchronization scheduling leads to considerably less cache divergence over the more conventional approach in which the cache unilaterally schedules refreshes. There are several interesting avenues for future work, discussed in [20].

Acknowledgments

We thank Mike Franklin, Hector Garcia-Molina, Rajeev Motwani, Mema Roussopoulos, and Nick Roussopoulos for their helpful discussions and feedback. We also thank Junghoo Cho for providing useful information regarding the CGM algorithm.

[4] More generally, L_i could represent the end-to-end latency between the time a real-world event occurs, triggering a change to the source data, and the time an application reading data from the cache sees the change.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying update streams in a soft real-time database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 245–256, San Jose, California, May 1995.

[2] B. Adelberg, B. Kao, and H. Garcia-Molina. Database support for efficiently maintaining derived data. In *Proceedings of the International Conference on Extending Database Technology*, pages 223–240, Avignon, France, Mar. 1996.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the Seventh International World Wide Web Conference*, Brisbane, Australia, Apr. 1998.

[4] G. S. Brodal, J. L. Träff, and C. D. Zaroliagis. A parallel priority queue with constant time operations. *Journal of Parallel and Distributed Computing*, 49(1):4–21, Feb. 1998.

[5] M. Cherniack, M. J. Franklin, and S. Zdonik. Expressing user profiles for data recharging. *IEEE Personal Communications: Special Issue on Pervasive Computing*, 8(4):32–38, Aug. 2001.

[6] J. Cho and H. Garcia-Molina. Estimating frequency of change. Technical report, Stanford University Computer Science Department, 2000. http://dbpubs.stanford.edu/pub/2000-4.

[7] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 117–128, Dallas, Texas, May 2000.

[8] E. Cohen and H. Kaplan. Refreshment policies for web content caches. In *Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, Anchorage, Alaska, Apr. 2001.

[9] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy. Adaptive push-pull: Disseminating dynamic Web data. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, China, May 2001.

[10] L. Do, P. Ram, and P. Drew. The need for distributed asynchronous transactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 534–535, Philadelphia, Pennsylvania, June 1999.

[11] T. Dorcey. CU-SeeMe desktop videoconferencing software. *Connexions*, 9(3), Mar. 1995.

[12] D. Estrin, L. Girod, G. Pottie, and M. Srivastava. Instrumenting the world with wireless sensor networks. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP 2001)*, Salt Lake City, Utah, May 2001.

[13] A. Gal and J. Eckstein. Managing periodically updated data in relational databases: A stochastic modeling approach. *Journal of the ACM (to appear)*, 2002.

[14] R. A. Golding and D. D. E. Long. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical report UCSC-CRL-93-09, Computer and Information Sciences Board, University of California, Santa Cruz, 1993.

[15] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Network Monitoring (MobiCom 99)*, pages 271–278, Seattle, Washington, Aug. 1999.

[16] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the Web. In *Proceedings of the Twenty-Seventh International Conference on Very Large Data Bases*, pages 391–400, Rome, Italy, Sept. 2001.

[17] M. J. McPhaden. Tropical Atmosphere Ocean Project, Pacific Marine Environmental Laboratory, 2001. http://www.pmel.noaa.gov/tao/.

[18] C. Olston, B. T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 355–366, Santa Barbara, California, May 2001.

[19] C. Olston and J. Widom. Offering a precision-performance tradeoff for aggregation queries over replicated data. In *Proceedings of the Twenty-Sixth International Conference on Very Large Data Bases*, pages 144–155, Cairo, Egypt, Sept. 2000.

[20] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. Technical report, Stanford University Computer Science Department, 2001. http://dbpubs.stanford.edu/pub/2001-43.

[21] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):551–558, May 2000.

[22] C. Pu and A. Leff. Replica control in distributed systems: An asynchronous approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–386, Denver, Colorado, May 1991.

[23] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, 1(2):199–226, 1993.

[24] G. Salton and C. S. Yang. On the specification of term values in automatic indexing. *Journal of Documentation*, 29:351–372, 1973.

[25] P. Sanders. Randomized priority queues for fast parallel access. *Journal of Parallel and Distributed Computing*, 49(1):86–97, Feb. 1998.

[26] J. Stewart. *Calculus: Early Transcendentals, Second Edition*. Brooks/Cole, 1991.

[27] B. Urgaonkar, A. G. Ninan, M. S. Raunak, P. Shenoy, and K. Ramamritham. Maintaining mutual consistency for cached Web objects. In *Proceedings of the Twenty-First International Conference on Distributed Computing Systems*, Phoenix, Arizona, Apr. 2001.

[28] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, California, Oct. 2000.