Adaptive Pull-Based Policies for Wide Area Data Delivery*

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

Avigdor Gal
Technion - IIT
Haifa 32000 Israel
avigal@ie.technion.ac.il

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

Abstract

Wide area data delivery requires timely propagation of up-to-date information to thousands of clients over a wide area network. Applications include web caching, RSS source monitoring, and email access via a mobile network. Data sources vary widely in their update patterns and may experience different update rates at different times or unexpected changes to update patterns. Traditional data delivery solutions are either push-based, which requires servers to push updates to clients, or pull-based, which require clients to check for updates at servers. While push-based solutions ensure timely data delivery, they are not always feasible to implement and may not scale to a large number of clients. In this paper we present adaptive pull-based policies that can reduce the overhead of contacting remote servers compared to existing pull-based policies. We model updates to data sources using update histories, and present novel history-based policies to estimate when updates occur. We present a set of architectures to enable rapid deployment of the proposed policies. We develop adaptive policies to handle changes in update patterns, and present two examples of such policies. Extensive experimental evaluation using three data traces from diverse applications shows that history-based policies can reduce contact between clients and servers by up to 60% compared to existing pull-based policies while providing a comparable level of data freshness. Our adaptive policies are further shown to dominate individual history based policies.

*This research is partially supported by NSF Grants IIS0205489,IIS0219909 and EIA0130422.

1 Introduction

Wide area data delivery involves the timely transfer of data from servers to tens of thousands of clients over a wide area network. Wide area applications are often characterized by frequent updates that must be disseminated to clients in a timely manner. Examples include cache managers maintaining local copies of web pages, clients interacting with a mail server, delivery of data from RSS news feeds or the stock market, clients monitoring online auctions, etc. A challenge to timely update propagation is that most data sources are inherently pull-based and require clients to contact them to check for updates. While push-based data sources and services exist, e.g., BlackBerry [23], these can support only a limited number of clients and data sources. The majority of servers are either unable or unwilling to support push-based data delivery; thus, applications that access these servers must rely on pull-based delivery.

An important challenge to effective pull-based data delivery is minimizing the number of contacts between clients and servers while satisfying client freshness requirements. Minimizing the number of client requests to servers is important for several reasons. First, excessive client requests waste network bandwidth. Reducing bandwidth consumption is particularly important in low-bandwidth wireless environments, but is important on fixed wireless networks as well. Second, excessive client requests overload remote servers. Finally, contacting remote servers adds latency to client requests, and should only be done when local data copies are not sufficiently fresh. To minimize requests to servers, clients require reasonable estimates of when updates will occur.

Predicting updates to remote servers is non-trivial. Update patterns at a source may vary depending on many factors including time of day and day of the week, and sources may experience unexpected changes in update rates. To date, many heuristics to estimate update probability have been proposed, ranging from estimating a constant lifetime for all objects to modeling update probability based on the past update rate of an object (we discuss these in detail in Section 2). Such heuristics have proven effective for applications such as caching web pages that change relatively infrequently or synchronizing a large collection of objects. They may not be appropriate for wide area applications where clients may have specific requirements for timely data delivery and where data delivery must adapt to changes in the pattern of updates.

We present several examples of pull-based wide area applications and briefly discuss the challenges they present. Our first example is a client who monitors updates to one or more web sources of interest, such as sources that provide data on sports scores, stock quotes, or auctions. While ideally these sources could push a stream of updates to clients, in practice this is not always feasible. Thus, clients must poll sources to be notified of relevant updates. However, continuous polling strains network resources for both clients and servers and is clearly not a feasible solution. Instead, clients must intermittently poll sources to check for updates [12, 38, 39]. Further, update frequencies may vary at different times of the day or days of the week, and may change due to external events, e.g., an increase in the frequency of bids near the end of an auction. To ensure timely propagation of updates without wasting excessive network resources, a reasonable model of update patterns at remote sources and the ability to adapt to changes in these patterns is needed.

A second example is a web cache such as a client browser cache or a client-side proxy cache that serves multiple clients within an organization. These caches maintain local copies of popular objects to serve client requests, and can considerably reduce both network latency and bandwidth consumption. However, these local copies become stale as objects are updated at remote servers. To ensure freshness, caches typically use a Time-to-Live (TTL) for each object, which is either defined by a server (hereafter referred to as *server-defined* TTL) or estimated by the cache as a function of the object's last update [7, 22] (*adaptive* TTL). Any object that is requested after its TTL (either server-defined or adaptive) expires must be validated at a remote server to check for updates before it is delivered to the client. In the remainder of this paper we use the term TTL to refer to adaptive TTL.

While web caches typically have a high bandwidth connection to the Internet, validating objects that have not actually changed adds a considerable amount of unnecessary overhead to requests and reduces the benefits of caching. Research reported in [15] shows that in web caching as many as 30-50% of cache hits result in unnecessary validations where the object has not actually changed (i.e., freshness misses), and the latency of these validations is often comparable to that of a cache miss. Thus, improving the ability of pull-based policies to estimate the freshness of cached objects could potentially double the number of requests served from a cache while reducing the latency of a significant percentage of requests.

A third example is a client who uses a mobile PDA to access data such as web objects and email. The client needs to periodically contact the remote sources to refresh their mailbox and check for updates to local object copies. Unlike in the proxy caching example above, in this case each contact with the server is costly in terms of both battery power and bandwidth. Thus, continuous polling is impossible in this case. However, it is important for clients to receive update information in a timely manner. Therefore, efficient techniques to estimate when updates occur could significantly reduce both power and bandwidth consumption while improving the timeliness of data delivery to clients. We note that BlackBerry [23] provides a push-based solution to this problem, but this may not be feasible for all clients and applications.

To summarize, minimizing the number of client requests to remote servers is important for all of the above applications. Further, accurately modeling and estimating updates to objects can significantly reduce the overhead of contacting servers while still meeting the freshness requirements of clients. In this paper, we address the following (informal) problem: We are given a set of objects, a model of updates to these objects, a stream of requests for these objects, and possible freshness and latency constraints. The objective is to minimize the total number of requests to the servers while meeting the freshness and latency constraints. We formally define the problem in Section 3.2.

Challenges to solving the problem above include the following:

- Models of updates to objects at remote sources.
- Policies that use these models to estimate when objects are updated and to determine when to contact sources to check for updates.
- Architectures to efficiently support and deploy these policies.
- Adapting to changes in update patterns at sources.

In this paper, we present adaptive pull-based policies to address the above challenges. Our goal is to enable pull-based policies to provide sufficiently recent data for applications while reducing communication overhead and power consumption. Our policies can be implemented with minimal changes to clients and servers, and can detect and adapt to changes in update patterns at sources. While existing works in web caching ([10, 22, 32]) and synchronizing collections of objects ([12, 9, 13, 38, 39]) have proposed pull-based policies (see Section 2 for detailed discussion) including policies that use history, our work makes several novel contributions to be discussed next. In addition, we know of no other prior work in pull-based data delivery that explicitly aims to minimize the number of client requests to remote servers while satisfying client requirements for latency and recency.

In Section 3, we formally define the problem addressed in this paper. We present our model of pull-based data delivery and present two formulations of the problem. The first formulation aims to minimize the latency of requests but allows requested objects to be downloaded from remote servers when a cached copy does not meet client freshness constraints. This formulation is well-suited to caching applications such as refreshing objects in a web cache in response to client requests. The second formulation assumes hard latency constraints and assumes all requests are served from a cache while objects are refreshed in the background. This formulation is well-suited to prefetching applications such as periodically refreshing a client's mailbox.

In Section 4, we present a model for update history and introduce the notion of bursts. The update model is based on a cyclic stochastic model (which means one can predict the future based on past behavior). The model can be extended with "bursts" or deviations from the cyclic history; we refer to the presence of bursts as an acyclic history. We utilize the work in [20] to model history as a repetitive piecewise constant Poisson process, which is a step beyond the homogeneous models that are currently utilized in history modeling (e.g., [12, 13]). This history may either be the history of an individual object, or may be aggregated over multiple objects.

We present a set of flexible and scalable architectures to support history-based policies in Section 5. These architectures differ in the overhead required by both servers and clients, allowing our policies to be supported by clients and servers with varying capabilities. The first architecture, client-side, proposes piggybacking a compact representation of update histories on server response, thus allowing the client the flexibility of computing its policies independently. This requires minimal changes to a server and enables servers to scale to a large number of clients, but consumes power and storage at clients. For clients with limited storage or processing power (such as mobile clients), we propose a server-side architecture in which the server computes the estimated time of an object's next update, similar to server-defined TTL yet using a policy that is based on history. This value is then sent to the client very much the same way server-defined TTL is sent nowadays. We note that our proposed architectures do not require changing underlying network protocols. Further, while both of these architectures do require cooperation from remote servers, the amount of cooperation required is far less than for typical push-based architectures.

We develop three pull-based policies in Section 6. The well-known adaptive TTL policy [22] is presented as a degenerate case of a history-based policy, utilizing the most recent update to predict future updates. In addition, we present two more policies, labeled *IndHist* and *AggHist*, that base their prediction on the repetitive nature of update history, utilizing more updates than TTL in generating the history. The two policies differ in the way they utilize update histories. IndHist bases its prediction for a given object solely on the updates to that object while AggHist utilizes the updates of a set of similar objects in prediction. IndHist may be more accurate than AggHist for "nicely-behaved" objects, but has the disadvantage of over-fitting individual object histories.

Section 8 offers two examples of adaptive policies that take advantage of multiple pull-based policies to improve performance. The first replaces IndHist with AggHist for objects that do not yet have sufficiently recorded updates to use IndHist. The second adaptive policy identifies bursts and switches from IndHist to adaptive TTL [22], to avoid the use of an incorrect history model until after the burst is over. A client determines which policy to use, based on performance. Therefore, the detection of a burst will result in switching from IndHist to TTL. Once a burst has subdued, the client may switch back to IndHist. In both cases, we show empirically that adaptive policies perform better than any of the three policies executed independently, thus providing a mechanism for robust decision making in the face of continuous changes in update patterns.

We have conducted extensive experiments to support our analysis, using real-world traces from various domains, including electronic bulletin boards (DB-WORLD), mail and web server applications (World Cup 1998). Our findings are reported in sections 7 and 8. Our results show that history based policies can reduce contact between clients and servers by up to 60% compared to TTL while providing a comparable level of data freshness. We show that for objects with cyclic history, history-based policies dominate TTL. This is because TTL assumes that objects updated recently are more likely to be updated in the near future, which does not hold for cyclic objects, and thus performs many more unnecessary validations than history-based policies. We further show that for objects with acyclic history that exhibits bursts, history-based policies, while not completely accurate, are sufficiently robust to provide useful predictions of updates.

Our experiments also demonstrate that our adaptive policies can select the best policy to match the update behavior of an object and they adapt to changing behavior. One adaptive policy exploits both individual and aggregate behavior by alternating IndHist and AggHist. A second adaptive policy alternates IndHist with TTL; TTL is exploited when there are bursts, whereas IndHist is utilized in non-bursty periods. Thus, the adaptive policies dominate any of the individual history-based policies.

To summarize, our main contributions are as follows:

- We formalize the model of update history, for either an individual object or aggregated over multiple objects, as a cyclic stochastic model with possible bursts.
- We present a set of offline optimal history-based policies to estimate when updates occur at sources and show empirically that using history-based policies can significantly reduce the communication overhead between clients and servers.
- We demonstrate the flexibility of our approach by suggesting a set of architectures to implement support for our policies.
- We propose a set of adaptive policies to choose among available policies to proactively handle changes to a source's update patterns. Our empirical results establish the dominance of the proposed adaptive policies over individual policies.

2 Related Work

The problem of maintaining freshness of local data copies has been studied in many contexts, including web caching, web crawling, and databases. All of this research shares our goal of maintaining data freshness while making optimal use of available resources. Some approaches are push-based, providing guarantees with respect to data freshness at a possibly considerable overhead at the server. Other approaches are pull-based, reducing overhead at servers, with limited effectiveness in estimating object freshness. Some existing pull-based policies use update histories to estimate the freshness of data copies, however, their use of histories differs considerably from the policies proposed in this paper.

In this section we survey related work in pull-based and push-based freshness policies, and briefly discuss the tradeoffs of each. We also discuss existing techniques to model history. In Section 2.1 we present existing pull-based policies for wide area applications and discuss their strengths and weaknesses. We compare their overhead, accuracy, and use of update histories to the techniques presented in this paper. In Section 2.2 we survey push-based freshness policies for wide area applications and compare them in terms of server overhead and the degree of freshness guaranteed. We discuss some tradeoffs between push-based and pull-based data delivery. Finally, in Section 2.3 we discuss caching and freshness policies in other areas such as distributed databases, materialized views and filesystems.

2.1 Pull-Based Freshness Policies

Pull-based freshness policies require clients to contact servers to check for updates to objects. Such policies have been proposed in many contexts such as web caching and synchronizing collections of objects, e.g., web crawlers. While this research shares our goal of maintaining freshness of data copies, these policies have many differences from those proposed in this paper. We discuss these differences below.

2.1.1 Web Caching

There has been much research in the web caching community on pull-based freshness policies. These policies typically rely on simple heuristics to estimate the freshness of a cached object, for example estimating freshness as a function of the last time the object was modified. The heuristics to estimate freshness are typically less accurate than using more complete history information which can model variations in an object's update frequency at different times.

A widely used pull-based policy is to assign each object a Time-to-Live (TTL) [10, 22], and validate any cached object whose TTL has expired. The TTL value is typically either a fixed value provided by a server (server-defined) or is estimated as a function of the time that an object was last modified (adaptive TTL [22]). (In the remainder of this paper the term TTL refers to adaptive TTL, defined formally in Section 6.1, unless otherwise noted.) More recently, work in [31] aims to improve upon this by estimating TTL values based on the probability that an object will be updated within a certain period of time. The authors have suggested tuning K, a shifting window that consists of the number of most recent updates to be utilized in computing update events arrival rate. This approach may capture local arrival rate values, as they vary over time, yet it is more error prone in the borderline, where the arrival rate either sharply increases or decreases. The technique suggested in [31] is similar to the First Arrival policy suggested in [20]. In our research we have adopted a different technique, also suggested in [20], which considers the expected number of arrivals rather than the probability of an update. To avoid misestimation when histories have changed or have not yet stabilized, we also suggest in this work an adaptive approach, in which a client may switch between policies whenever update patterns change.

The policies to estimate object freshness, as proposed in this paper, are based on stochastic analysis of update history. Several works, including [12, 20, 31] have suggested modeling updates as a Poisson model. Work in [12, 31] assumes a model that is homogeneous over time, while our proposed model assumes a time varying update intensity, which was shown to work better in [20].

In prior work we have considered the importance of estimating the number of updates in a given interval. In [6] we proposed the use of client profiles, where a client can define its tolerance towards stale data in terms of the number of updates to an object since it was last refreshed. It was shown in [20] that clients that are more tolerant to stale data suffer less latency in processing their data requests. In this paper we introduce a third aspect to varying levels of tolerance towards staleness by offering compact methods for improving client estimation of the number of updates. Equipped with improved methods to estimate the number of updates to an object, clients can meet the recency requirements of their applications while reducing unnecessary contacts with remote servers.

Research reported in [15] considers pre-validation policies to validate cached objects whose TTLs have expired before clients request them; this technique has been shown to reduce the client-perceived latency caused by unnecessary validations (freshness misses). These policies can reduce the number of validations in response to client requests, however, they *increase the total number of contacts with the server* because objects must be validated offline. In contrast, the policies we present in sections 7 and 8 can reduce the number of unnecessary validations *and* total contacts with servers while providing comparable recency to TTL.

Piggybacking to improve freshness was proposed in [28]. In this work, clients piggyback a list of potentially stale cached objects when they contact a server. Servers piggyback the subset of those objects that have been updated on their responses. Our notion of piggybacking is completely different. In one of our proposed architectures (see Section 5), servers piggyback the history of updates to clients to provide clients with a better estimate of when updates occur. This approach slightly increases the size of a request but can reduce the number of contacts with the server and reduce latencies.

2.1.2 Synchronizing Collections

Pull-based freshness has also been addressed in the context of synchronizing a large collection of objects, e.g., web crawlers [9, 12, 13, 14]. These works propose policies for prefetching objects from remote sources to maximize the freshness of objects in the cache. The goal is to refresh a collection of objects offline, rather than handle client requests online; hence, these policies are not always appropriate for all applications as discussed below. Research reported in [38, 39] presents pull-based policies to monitor changes to web sources, but does not model update histories. We describe all of this work below.

Research reported in [13] detects updates to objects by periodically contacting servers, so it may not have complete update histories. Further, this approach uses available history information to estimate only the *average update frequency* of an object, thus ignoring the nonhomegeneity of update patterns. Thus, this use of update history differs from what we propose in this paper.

There are several important differences between the synchronization problem addressed in [9, 12, 13, 14] and the problem of estimating the freshness of a cached object on-demand that we study in this paper. In web crawling, the goal is to develop crawling strategies to maximize the average freshness of a set of objects over a period of time subject to resource constraints, e.g., bandwidth. These strategies typically determine how often to check for updates based on its *average* update frequency. They do not attempt to estimate when an object is most likely to be updated. This is appropriate for applications such as web crawling because some degree of staleness is acceptable and it simplifies implementation.

In contrast, this paper focuses on using history to estimate the freshness of objects for applications where capturing updates in a timely manner is critical. This includes on-demand applications such as web browsing, email, and online auctions, as well as applications such as source monitoring that must capture all relevant updates to a source. In these cases, using the average update frequency to estimate the freshness of an object is insufficiently accurate and may not meet the preferences of the client (i.e., the object may be too stale or important updates may not be captured). Therefore, in this paper we propose more sophisticated models to estimate the freshness of a cached object.

The WIC algorithm [38] converts pull-based data sources to push-based streams by periodically checking sources for updates. The algorithm is parameterized to allow clients to control the tradeoff between *timeliness* (being notified of updates promptly) and *completeness* (being notified of all changes to an object) when bandwidth is limited. At each time interval the algorithm chooses the objects to refresh based on both client preferences and the probability of updates to an object. This algorithm is useful for many wide area applications such as online auctions or archiving web sources. However, the algorithm does not consider how to determine the probability of an update to an object, which is an important aspect of any pull-based policy.

CAM [39] also proposes pull-based monitoring of web sources with the goal of capturing as many updates as possible. This work estimates the probability of updates by probing sources at frequent intervals during a tracking phase, and using these statistics to determine the change frequency of each page. However, CAM does not explicitly model time-varying update frequencies to sources and cannot easily adapt to bursts.

2.2 Push-based policies

Push-based data delivery guarantees that relevant updates are delivered to clients in a timely manner and is useful for many applications. However, push-based delivery may not be cost effective or feasible in all situations. The tradeoffs between push-based and pull-based data delivery have been studied extensively, (see, for example, [19] for a detailed discussion). Some scenarios are well suited to push, for example a small group of clients who need to be notified of updates in a timely manner. In this case push-based delivery could eliminate the need for clients to frequently poll servers, thereby reducing server load. We note that just as push-based technologies can benefit from commonalities among client requests and reduce server loads, our proposed pull-based approaches can exploit similarities at the proxy level. Thus, a proxy could perform pull-based monitoring of sources for a large group of clients, further reducing the load at servers.

Many push-based technologies are currently in use, e.g., BlackBerry and JMS messaging. However, these solutions require significant changes to servers. In addition, they require servers to maintain information on every client's preferences, which may limit scalability. In contrast, our proposed solutions require minimal modifications to server implementation, do not require servers to store any client information, and can be deployed incrementally as discussed in Section 5.

Finally, we note that many existing “push-based” publish/subscribe services (e.g., RSS feeds) are actually implemented using periodic pull. The solutions proposed in this paper can improve the effectiveness of such techniques by reducing the number of contacts with servers in many cases. Thus, we believe our proposed solutions can improve both pure pull-based delivery as well as many publish/subscribe solutions.

We discuss existing push-based policies in the context of caching static and dynamic web content, and research in caching approximate values below.

2.2.1 Static web content

Push-based policies for static web content are considered in [16, 32]. These policies may be useful for keeping data in client-side web caches up to date, but impose a large overhead on servers and may not scale well to a large number of clients. Research reported in [32] compares TTL to a push-based policy and shows that the push-based policy does not consume significantly more bandwidth than TTL in many cases. This comes as no surprise, given that push-based policies only contact clients when an object actually changes, thus unlike TTL there are no freshness misses. However, push-based policies also require additional overhead for servers, and many servers may be unwilling or unable to implement it. Research reported in [16] proposes an adaptive push-pull scheme where servers can adaptively push updates to some clients and require others to use a pull-based policy. The server does not provide any update history information to clients. The server determines which clients use push and which must use pull-based on both the server capacity and the preferences of the clients. This improves scalability but may not meet client needs when server capacity is limited.

2.2.2 Caching Dynamic Content

Many works, including [2, 8, 11, 29, 45, 46] consider push-based consistency in the context of caching dynamic web content. These solutions are well-suited to improving the performance of dynamic web sites, but do not generalize well to wide area networks. This is because caching dynamic content requires a high degree of cooperation between caches and servers. Caches must store database tuples or page components as well as scripts to generate web pages in response to client requests. These caches are typically controlled by servers rather than clients. Since the number of server-side caches is much smaller than the number of clients accessing the pages, scaling to a large number of clients is not a concern and pushing updates to the caches is feasible.

Research reported in [8, 11, 29, 46] proposes techniques to determine which pages the server should invalidate when updates occur to the underlying database.

The goal is determining when to propagate updates to cached data, assuming full knowledge of updates to the underlying database. Push-based consistency in also considered in [45]. This research evaluates the effects of push-based consistency on a server workload and shows how servers can limit the amount of information they can store without significantly impacting data consistency. However, servers still must store information about clients, which our solution does not require. Finally, research reported in [2] proposes techniques to cache database tuples to answer queries. Servers send periodic refresh messages to notify caches of tuples that have changed, which guarantees that cached data is consistent with a past database state within some constant time limit. In contrast, our work provides a pull-based approach that allows clients to contact servers to maintain their desired level of freshness.

2.2.3 Caching Approximate Values

There is also work in push-based consistency that allows cached objects to deviate from objects at the remote server [1, 26, 36, 37]. These techniques can save bandwidth by reducing the amount of communication between servers and clients. However, as with other push-based policies, servers need to store the inventory of objects in a client's cache. In addition, servers need to store the client's tolerance towards deviation of object values in the cache from that stored on the server side. Therefore, approximate caching solutions may impose excessive overhead on servers when there is a large number of clients, and are better suited to applications with a small number of clients and servers.

Several recent works in approximate caching aim to provide greater flexibility to both clients and servers and to reduce communication overhead when bandwidth is limited. Research reported in [36] presents an adaptive policy for servers to bound the deviation between objects at servers and client copies. For each object, servers set a window bounding its precision. When the window exceeds a client's preferred degree of precision, the client must request a fresh object from the server. When the server value falls outside this window, the server must push the update to clients. The authors propose techniques for servers to automatically adjust the window size to minimize contacts between clients and servers. This approach is an effective way to reduce communications between clients and servers, but requires considerable storage and computational overhead at the server.

In [37] the authors consider synchronizing a set of cached objects when both cache-side bandwidth and server-side bandwidth may be limited. Servers cooperate with clients when determining which objects to refresh, and both clients and servers can adjust the deviation of a cached object when bandwidth is limited. This solution is appropriate for applications such as sensor networks where there is a finite number of clients and bandwidth is limited, but would not scale well to wide area networks.

2.3 Caching in other contexts

For completeness, we briefly present caching in other contexts, namely materialized views and distributed databases and filesystems.

Research in the area of materialized views, e.g., [4, 21, 24] precomputes answers to database queries to reduce query execution time. Queries can be answered using precomputed views, which is faster than querying the underlying database. As in web caching, a key challenge in materialized view research is keeping the views fresh when updates are made to the underlying database. However, the challenge is to reduce the computational overhead of recomputing views, rather than to reduce network latency to clients. This research typically assumes full knowledge of updates to the underlying database, i.e., push-based consistency. Therefore, they do not address the issue of how often to check for updates. Further, since a single database has a finite number of materialized views, scalability to thousands of clients is not a concern.

There is a considerable amount of research in caching and cache consistency in distributed filesystems [25, 34], client-server databases [18], and distributed shared memory [27]. In all of these areas, caching is typically performed on a relatively small number of machines connected by a high bandwidth network. Thus, this research does not consider the challenges of scaling to a large number of clients and limiting bandwidth consumption on wide area networks that we consider in our work. Further, in distributed databases and filesystems clients can both read and write to cached copies. Thus, this research typically requires concurrency control mechanisms with a high communication overhead. In contrast, in wide area applications updates occur only at the server and client copies are read-only.

3 Problem Definition

We start with describing a pull-based model for wide area data delivery and explicitly state our assumptions. We then present the optimization problem.

3.1 A Pull-Based Model

In a pull-based model, clients request data objects from servers. In this paper, we use the term *client* to refer to either an individual user or a proxy, e.g., a shared cache or a publish/subscribe service that pulls from multiple sources. We focus on objects that can only be pulled, and therefore, clients need to develop policies to determine when to pull data. Consider the time period $[0, \mathcal{T}]$, and let $\{p_i\}_{i=1}^{n}$ represent the times the client probes the server for an object. At each probe i, the client is synchronized with the state of the server at the time of the probe p_i; the information is available immediately. At subsequent probes, the client is informed of the up-to-date state at the server, e.g., $U(p_i, p_{i+1})$, the number of updates during the interval $(p_i, p_{i+1}]$. We define $p_0 = 0$, and require that $0 \leq p_1 \leq p_2 \leq \cdots \leq p_n \leq \mathcal{T}$. We note that we assume that all updates are of equal importance to the client. Research in [38] introduces other parameters such as urgency which differentiate the importance of updates.

A client can specify constraints or objectives for optimization along the following three dimensions:

Freshness dimension: The freshness dimension indicates the tolerance of a client towards obsolescent data. We assume that this dimension is specified in terms of a threshold on the expected number of updates, since a client can only estimate the number of updates. Recall that a client can compute the expected number of updates, given some update model, as discussed in Section 4. Given a threshold θ, the freshness constraint is met if for two consecutive probes at times p_i and p_{i+1},

$$\mathrm{E}\left[U(p_i, p_{i+1})\right] \leq \theta$$

This means that a user can tolerate up to θ expected updates that occur at the server side during the interval $(p_i, p_{i+1}]$.

Latency dimension: The latency dimension indicates the tolerance of a client when it comes to waiting for the data. On the extreme, zero latency tolerance results in a latency constraint that requires the data to be present whenever requested. Such a constraint, marked $L = 0$, is typically handled by pre-fetching methods to ensure availability. As an example, one may consider the availability of emails in a mailbox upon accessing it. In less strict scenarios, a cache can determine whether to service a client from the cache (assumed to have zero latency) or to request a fresh copy from the server with some upperbound on the latency. In our research, we simplify this constraint; a client is willing to accept indefinite latency, marked $L = \infty$. An example for the latter may be that of web browsing. Optimizing for minimum latency has been handled elsewhere [6].

Resource dimension: The resource dimension indicates the amount of resources a client may devote to pull data. For example, a politeness constraint [17] sets an upperbound on the number of pull requests to a single source in a time interval. As another example, consider bandwidth constraints [38] which considers an upperbound on the total number of pull requests to all servers.

Each of the above dimensions may be either a constraint or a target of optimization. For example, in [17], the resource dimension is considered to be a hard constraint, while the freshness dimension is the target of optimization. The latency dimension was not considered at all.

3.2 Optimization Problem

Let $r = \{r_j\}_{j=1}^{m}$ represent a set of client requests. Let $p = \{p_i\}_{i=1}^{n}$ represent the (times of) probes to the server. The goal of our optimization is to minimize

n, the total number of probes to the server, subject to freshness and latency constraints. The freshness dimension is considered a hard constraint. For the latency dimension, we consider two variants. In the first, latency is not a hard constraint and we assume that client requests may be satisfied either from the cache or from the server. By minimizing n, the number of probes to the server, we also minimize latency. For this variant, since some or all of the client requests may be served from the cache, it follows that the (times of the) probes are synchronized with the client requests, and the set of probes $p \subseteq r$ and $n \leq m$. In the second, latency is a hard constraint and all requests must be satisfied from the cache (Latency =0). The goal is to ensure that at all times, independent of when the actual client requests are made, cached objects satisfy client freshness constraints. This is needed since we assume we cannot predict when a client will request an object and all requests must be served from the cache.

Formally, we address the two following optimization problems:

$$\begin{aligned} &\min\ n \\ &\text{S.T. } \forall 1 \leq i \leq n, \mathrm{E}\left[U(p_i, p_{i+1})\right] \leq \theta, \\ &\quad L = \infty \end{aligned} \tag{1}$$

or

$$\begin{aligned} &\min\ n \\ &\text{S.T. } \forall 1 \leq i \leq n, \mathrm{E}\left[U(p_i, p_{i+1})\right] \leq \theta, \\ &\quad L = 0 \end{aligned} \tag{2}$$

Problem 1 is a *caching problem* and determines the relevance of current cache content with respect to a specific client request and its freshness requirement. Since the set of probes p is a subset of the set of requests r, this formulation ensures that for any request r_j where $p_i < r_j \leq p_{i+1}$, $\mathrm{E}[U(p_i, r_j)] \leq \theta$. Note that the case $(p_i < r_j < p_{i+1})$ corresponds to request r_j served from the cache with no probe to the server, while the case $(r_j = p_{i+1})$ corresponds to request r_j being served by a probe to the server. Problem 2 is a *prefetching problem*, in which the schedule of probes must ensure that user freshness constraints are always met by all objects in the cache.

In Section 6 we present two policies AggHist and IndHist; each can provide provide an optimal solution to problems 1 and 2. However, we note that our optimal solution only provides a schedule for the *expected number of updates*. When the actual behavior of the update pattern deviates from the expected number of updates, then these solution may no longer be optimal. Our experiments, presented in Section 7, further show that the solutions corresponding to the AggHist and IndHist policies exhibit better performance on real data traces, in comparison to existing solutions in the literature.

4 Modeling of Updates

Our research assumes that objects can only be pulled and therefore, we need to be able to estimate the freshness of a copy of an object. This, in turn, depends on our ability to model updates. Estimation typically uses knowledge such as past updates over some period of time. Both update frequency and the presence of a cyclic (i.e., repetitive over time) pattern in an update process can vary considerably for different objects. Some objects may be deterministically updated at the same time every day, some may experience completely unpredictable update patterns, and some may lie in between these two extremes. Objects that exhibit cyclic update patterns can be more easily modeled than objects whose update patterns vary randomly, and are therefore unpredictable.

In this section, we provide a theoretical framework for modeling updates. We start by defining update times as a stochastic process, which repeats itself, i.e., it is cyclic, while allowing time-based variations, which we denote as *bursts*. A history is then defined as the update arrival intensity over time. Using update history, we classify update instantiations based on how well they fit with the model, separating cyclic instantiations (which fit the model well) from acyclic instantiations (which fail to obey the model), the latter containing bursts.

Section 4.1 provides the basics of the update model. This section is based on [20] and is given here for completeness sake. We then provide a method for estimating model parameters using update instantiation (Section 4.2). In Section 4.3 we classify update instantiations according to the ability of some history to accurately model the instantiation. Throughout this section we illustrate our techniques using a real-world data trace (1998 World Cup website [3]). Details of the trace, as well as two other data traces are in Section 7.1.

4.1 A Model for Cyclic Updates

Models, in general, are an idealized representation of a process. To be useful, we wish to make accurate predictions regarding the timing of updates to objects. This section introduces a model, based on [20], for cyclic updates. The model assumes a compound nonhomogeneous Poisson process with a cyclic intensity function. Our decision to use a Poisson process stems from the well known fact that Poisson processes model a world where data updates are independent from one another. In data sources with widely distributed access, e.g., incoming e-mails, postings to newsgroups, or posting of orders from independent customers, such an independence assumption seems plausible.

We next describe the model in three steps. First, we describe a Poisson process with nonhomogeneous update intensity (which we denote *history*). We then generate the compound nonhomogeneous model. Finally, we provide the model for the cyclic intensity function and provide a compact representation of a history.

We summarize all the symbols used in this paper in Table 1. Note that some of these symbols will be introduced later in the paper and should be ignored for now.

$O = \{o_i\}_{i=1}^{n}$	a set of updateable objects
Υ	a stochastic process representing updates arrival
$\lambda(t)$	history (instantaneous intensity function)
$\vec{J}$	time subsets
$\vec{H} = (\vec{J}, \vec{\lambda})$	compact history representation
$\tau = \{t_j\}_{j=1}^{\infty}$	an instantiated sequence of update events
T	random variable, representing the inter-arrival time between consecutive update events
$t\_refresh$	time a cached object was last refreshed at the server
$t\_lastmod$	last known modification time of a cached object
$t\_request$	time of the latest client request for an object, for the caching problem
$t\_next\_refresh$	next time an object should be refreshed, for the prefetching problem
α	tuning parameter of TTL policy
θ	tuning parameter of history based policies
T_{ind}	threshold for Adaptive IndHist/AggHist policy
T_{burst}	threshold for Adaptive IndHist/TTL policy

Table 1: Summary of symbols and their meanings

Let $O = \{o_i\}_{i=1}^{n}$ be a set of n updatable objects (e.g., a web page) and let Υ be a nonhomogeneous Poisson process [41, 43] with an instantaneous intensity $\lambda : \Re \rightarrow [0, \infty)$, representing *update events* to O. The number of update events occurring in any interval $(s, f]$ is a Poisson random variable with expected value $\mathrm{E}\,[U(p_i, p_{i+1})] = \int_s^f \lambda(t)\,dt$. We call $\lambda(t)$ an *update history* (or simply *history*) of O.

To illustrate the concept of non-homogeneity, consider Figure 1. Figure 1(a) shows the changes to the intensity of an arrival rate over a period of one day, using a piecewise-constant model, as will be discussed shortly. Figure 1(b) provides a pictorial representation of a constant arrival rate (i.e., $\lambda(t)$ is equal to a constant $\lambda > 0$ for all t), as is the case with a homogeneous Poisson model. Figure 1(c) and Figure 1(d) demonstrate the accumulation of the expected numbers of updates over a period of one week. While the accumulation for the homogeneous Poisson model is linear over time (since $\int_s^f \lambda\,dt = \lambda \cdot (f - s)$), the accumulation in the nonhomogeneous case changes with fluctuations in the arrival intensity $\lambda(t)$.

The last component of the proposed model involves cyclic modeling of a history, i.e., the update intensity function. Such modeling assists in predicting future update events. Given some length of time Q, such as one day or one week, suppose that the intensity function $\lambda(t)$ repeats every Q time units, that is,

$$\lambda(t) = \lambda(t - Q\lfloor t/Q \rfloor) \tag{3}$$

for all $t \in \Re$. For $\lambda(t)$ that satisfy the constraint in Equation 3, we say that the history $\lambda(t)$ is *cyclic*.

As an example of the proposed cyclic model assume that the interval $[0, Q)$ is partitioned into a finite number of subsets $\vec{J} = J_1, \ldots, J_K$, with $\lambda(t)$ constant throughout each J_k, $k = 1, \ldots, K$. Each J_k is in turn composed of a finite number of half-open intervals of the form $[s, f)$. For instance, in Figure 1(a) Q is one day and $k = 7$. Table 2 shows the λ values for each subset. Such a process is dubbed *recurrent piecewise constant* (RPC) [20] and is a simplified

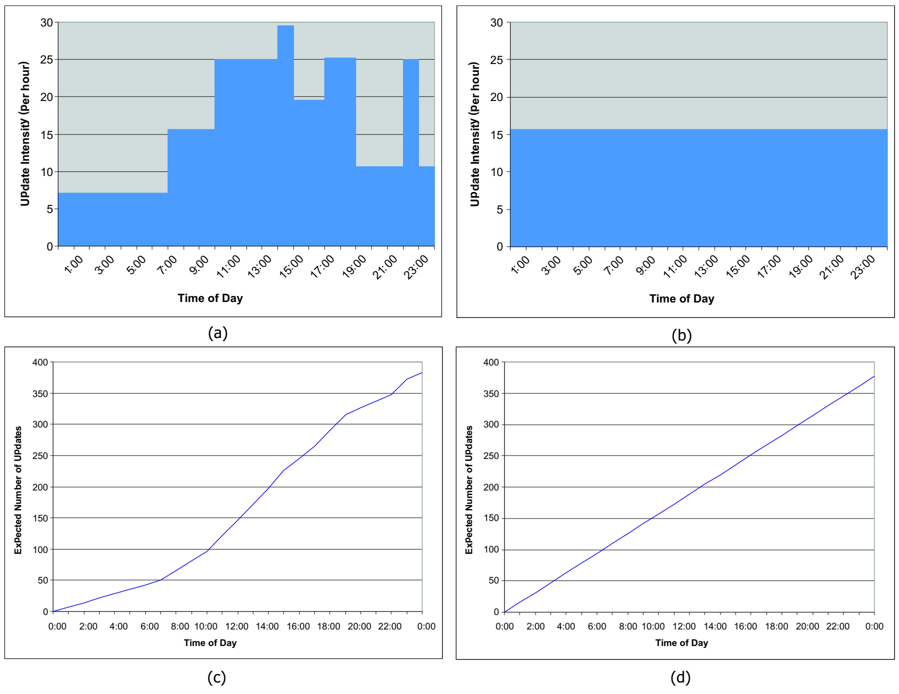


Figure 1: Homogeneous vs. nonhomogeneous Poisson models.

	Time	λ per hour
J_1	0-7	23.81
J_2	7-10	52.07
J_3	10-14,22-23	83.40
J_4	14-15	98.53
J_5	15-17	65.23
J_6	17-19	84.27
J_7	19-22,23-24	35.40

Table 2: An Update History ($\vec{H} = (\vec{T}, \vec{\lambda})$)

nonhomogeneous Poisson process, since $\lambda(t)$ is no longer continuous. Cyclic histories can be represented compactly as $\lambda(t)$ in the $[0, Q)$ interval. In the case of the RPC model above, a history is modeled as $(\vec{H} = (\vec{J}, \vec{\lambda}))$, a vector representing the effective λ value for each time fragment.

Let $T(i)$ $(1 \leq i \leq n)$ be n intervals, defined by the RPC model. Without loss of generality, assume that time s falls in interval 0 and time f falls in interval n. Therefore, whenever $n > 0$, we can compute $\mathrm{E}\left[U(p_i, p_{i+1})\right]$ to be

$$\mathrm{E}\left[U(p_i, p_{i+1})\right] =$$
$$(UB\left(T\left(0\right)\right) - s)\,\lambda\left(0\right) + \sum_{i=1}^{n-1}\left(\left(UB\left(T\left(i\right)\right) - LB\left(T\left(i\right)\right)\right)\lambda(i)\right) + \left(f - LB\left(T\left(n\right)\right)\right)\lambda(n) \qquad (4)$$

where $UB\left(T\left(i\right)\right)$ and $LB\left(T\left(i\right)\right)$ represent the upper bound and lower bound of $T\left(i\right)$, respectively.

Determining Q and $\vec{J}$ can be performed using a variety of methods, ranging from eye-balling to sophisticated statistical methods such as statistical process control (SPC) and change-point theory. In earlier experiments on a variety of data sets, we show that the statistical validity of the model was resilient to minor changes in the length of Q or the boundaries of the various intervals in $\vec{J}$.

4.2 Estimating Histories

Let $\tau = \{t_j\}_{j=1}^{\infty}$ be an infinite sequence of update times, representing an instantiated sequence of update events to O. Also, let Υ be a compound nonhomogeneous Poisson process with a cyclic history $\lambda(t)$ in the $[0, Q)$ interval, as described above. *Model fitting* is the process of estimating a history $\lambda(t)$ for any $t \geq 0$ using $\tau' = \{t_j\}_{j=1}^{k} \subset \tau$, partial instantiation of **consecutive** update times in $[0, Q)$. In this section we provide the details of the model fitting process for the RPC process. In what follows we assume, for simplicity sake, that for all $1 \leq j < k$ there is no $t_j < t < t_{j+1}$ such that $t \in \tau$ and $t \notin \tau'$. For estimating histories using incomplete update time sequences, the interested reader is referred to [13].

Recall that the RPC process is constructed from intervals in which λ is held constant, i.e., a homogeneous Poisson model. For such a model, it is well known that the inter-arrival time between any two consecutive update events is exponentially distributed with λ. Let T be a random variable, representing the inter-arrival time between two consecutive update events. $T \sim \exp(\lambda)$ and the expectation of T is

$$E\left[T\right] = \frac{1}{\lambda} \qquad (5)$$

Using Equation 5, we can now estimate λ by averaging inter-arrival times, as follows. Given $\tau^{'}$, one can compute the sequence of inter-arrival times $\bar{\tau}^{'} = \{t_j - t_{j-1}\}_{j=2}^{k}$. Given J_m, a subset of $[0, Q)$, and $\bar{\tau}_m^{'} = \{t_j - t_{j-1} | \{t_{j-1}, t_j\} \subseteq J_m\}$

one can estimate λ_m using Equation 5 as follows:

$$\hat{\lambda}_m = \frac{1}{\hat{E}[T]} = \frac{1}{avg_{t_j \in \tau'_m}(t_j - t_{j-1})}$$

since average is the unbiased estimator of $E[T]$.

An RPC model is essentially static over time intervals significantly longer than its period Q. Real-world processes may exhibit a combination of cyclic behavior and long-term changes, such as the gradual growth of a customer base or a sharp lasting rise in the popularity of a discussion topic. Thus, it would be useful to detect when the RPC model should be revised and refitted. The full methodology of model adaptation involves sequential decision making and process control techniques, which are beyond the scope of this paper, yet can be handled by advanced statistical mechanisms, as developed in *statistical process control* (SPC) and *change-point theory*. In these fields, problems of detection of changes (*monitoring*), estimation of the current process parameters (*filtering*), and identifying the change points and regimes (*segmentation*) have been tackled in numerous application areas, including industrial quality control, automatic fault detection in control dynamical systems (e.g., automatic pilots and robotics), and segmentation and pattern recognition of sound and image signals, e.g., [35, 44, 30, 40, 42, 33] and numerous references therein), and can be adopted in our case. The rich machinery supplied by this literature can be used to develop and implement a powerful, data-driven, and computationaly convenient methodology for monitoring the arrival rate λ and identifying its change points, regimes, and intensity levels. However, such a treatment is beyond the scope of this article.

4.3 Cyclic and Acyclic Update Instantiations

In this paper we propose adaptive pull-based policies based on history. In order to adapt, we need to be able to classify update instantiations and the policy which works best for each type of instantiation. Having introduced the cyclic update model, an obvious classification is to identify those instantiations that conform to the cyclic model and those that do not follow the model. We denote the former *cyclic* and the latter *acyclic*. One method of identifying cyclic vs. acyclic instantiations is by using hypothesis testing (e.g., the one provided in [20]) on a subset of the instantiation.

The usefulness of such a classification depends on the application domain. In some cases, a strict adherence to the model is needed to achieve good prediction. In other cases, even with acyclic instantiations reasonable prediction can be achieved. We use the World Cup 1998 update trace [3] to illustrate. The data set consists of all the requests made to the 1998 World Cup web site between April 30, 1998 and July 26, 1998. We report on 15 days of updates that occurred during June 10-25, 1998. In these 15 days there were 15,851 update times of a total of 5,507 objects. 255 of these were updated at least 10 times (172 cyclic, 83 acyclic). We describe our preparation and analysis of this trace in detail in Section 7.

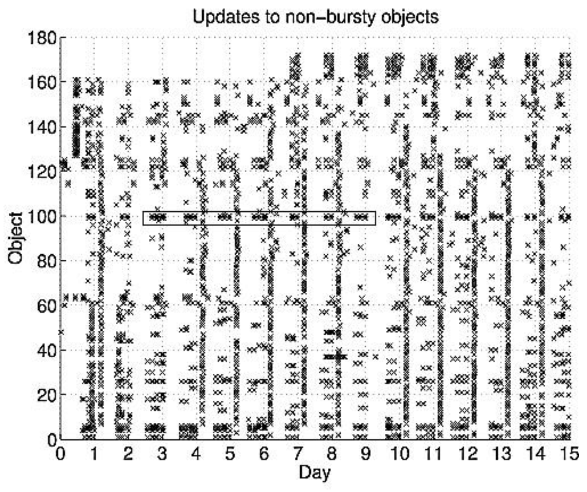


Figure 2: Cyclic update instantiations in the 1998 World Cup trace

Figure 2 plots the updates to cyclic objects that were updated at least 10 times in the 15-day trace period (172 objects total). In this figure, the x-axis is the time of day within a 15 day window, and each value on the y-axis represents a distinct object.

Figure 2 shows objects that exhibit close-to-cyclic behavior that is repeated daily. By "close-to-cyclic" we mean instantiations that seem to have some order, but cannot be validated statistically to fit with the proposed cyclic update model. In this case, objects were updated throughout the 15 days and it can be observed from Figure 2 that many of the objects are updated at the beginning of each day. As an example, the rectangle in the graph that spans from day 3 to day 9 in the graph highlights an object that experienced cyclic update patterns over this period. The object has a similar number of updates each day, and all updates occurred towards the end of the day. These objects may correspond to pages that provided daily updates on World Cup scores and events.

Acyclic instantiations can be modeled as a superposition of two separate instantiations, one of which is cyclic and the other generates a *burst* to the cyclic instantiation. Given an acyclic instantiation τ, modeled by a stochastic cyclic process Υ with an instantaneous intensity $\lambda : \Re \rightarrow [0, \infty)$, a burst occurs at time t if an update in τ at time t cannot be predicted by the history $\lambda(t)$. At the extreme, the cyclic instantiation is defined by $\lambda(t) = 0$ for all t, indicating no pattern. However, in other cases, it may become beneficial to identify the cyclic pattern of a acyclic instantiation and then identify bursts.

Modeling acyclic instantiations as a superposition of instantiations has several conceptual benefits. First, it separates a situation of temporary bursts from a permanent shift in the update model. The former has a temporary impact while the latter introduces a lasting change in the model. The use of superpositioning allow us to model bursts by adding short-term update model on top of an existing update model. Also, we have assumed that bursts are unpredictable. Therefore, we cannot apriori identify when a burst starts or how long it lasts. In future research we shall discuss extensions to the model, in which bursts occur in response to external events and can therefore be predicted. Again, using superpositioning, one can estimate the expected impact of a burst. Finally, superpositioning extends nicely to situations in which updates occur in static data sets, where no updates are expected. Modeling it as a superposition of a null update model (i.e., no updates) with a short-term burst of updates, avoids the nead to treat such situations separately.

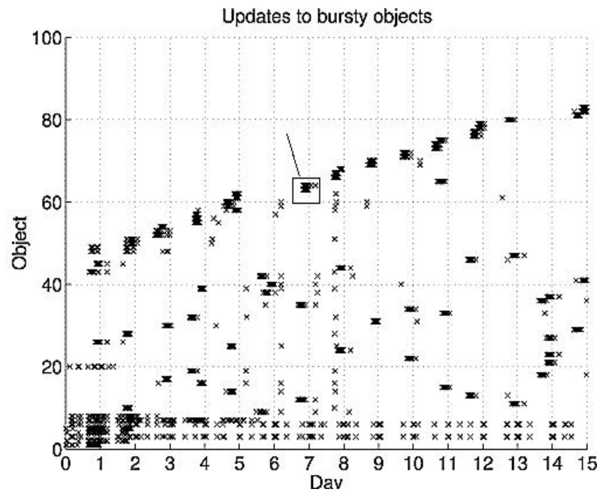


Figure 3: Acyclic update instantiations

Figure 3 shows acyclic update instantiations in the World Cup data trace with 10 or more updates (83 objects total). In this trace, acyclic instantiations have most of the updates on the same day, and few updates before or after the burst (this corresponds to the extreme case where the cyclic update instantiation is defined by $\lambda(t) = 0$ for all t). These instantiations may correspond to information on a specific World Cup event. The box indicates an example of a burst of updates. As an example, consider the score of a match. Many updates to the object occur on the day of the match, but few updates occur on other days.

Figure 4 illustrates an acyclic instantiation that is the result of superimposed burst over a cyclic instantiation. We generated this pattern synthetically by merging 55 cyclic objects from the World Cup trace with 55 bursty objects (we describe this in detail in Section 7.2.1). The arrow indicates an example of a bursty period.

Identifying bursts is far from being an easy task. The task of identifying bursts is different from the task of model adaptation mentioned above. Bursts

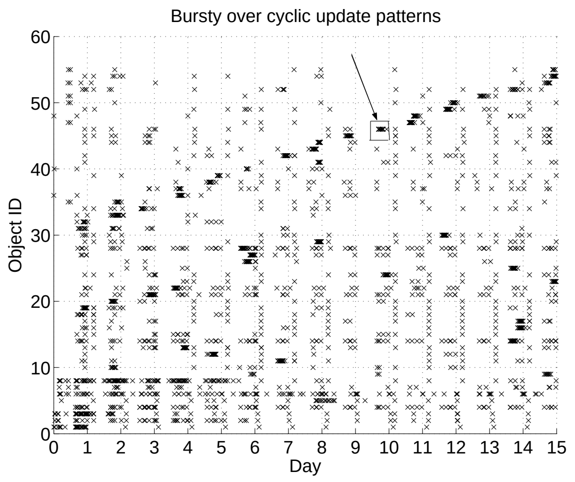


Figure 4: Bursty over cyclic update instantiations

may span a short time period, so statistically validated changes may not be useful here due to the relatively small amount of available data. Also, we build histories from non-deterministic data, and therefore minor deviations from what is expected according to a history $\vec{H}$ can be easily attributed to normal model variations rather than a burst. Therefore, we look in this paper into methods for identifying bursts. In Section 8.2.1 we explain the trade-offs in various methods for identifying bursts and justify our method of choice.

5 Architectures

This paper introduces several policies (to be presented in Section 6) that estimate the freshness of object copies using the update instantiation or history presented in the previous section. A challenge to implementing any such policy is determining where computation should be performed, i.e., at the client, server, or an intermediate proxy. In this section, we present two architectural approaches to implement pull-based freshness policies that use history. We note that both of these architectures require minor changes to servers, but fewer changes than support for push-based policies. These architectures are relevant for all of our motivating applications including web caching, source monitoring, and caching for mobile clients [5]. These alternatives differ by both the required storage and computational overhead for both clients and servers. Both architectures are suited to both the caching problem and the prefetching problem defined in Section 3.2. In the following discussion, the term *client* refers to either a cache on an individual client machine, (e.g., a mobile device) or a client-side proxy cache. The term *server* refers to either a server or a server-side proxy.

In the first architecture, *client-side*, the majority of the computation is done at the clients, while in the second architecture, *server-side*, the computation is done at the server. Both of these architectures are straightforward to implement by encoding the appropriate information in the HTTP headers of requests and responses. Support for either of these architectures requires only mutual agreement between the participating clients and servers and does not require any changes to the HTTP protocol. Servers and clients that do not support update histories will simply ignore the headers. Thus, both of these architectures can be deployed incrementally by any server or client that is willing to support history-based policies and will not affect any clients or servers that do not support such policies.

5.1 Client-side

The *client-side* architecture allows clients to maintain histories locally and use history as they see most appropriate for their applications. In this architecture, servers cooperate and piggyback history to their responses to client requests, and clients can choose how to use this information to estimate object freshness. This increases client flexibility at the cost of increased storage and computational overhead.

The client-side architecture works as follows:

1. The client requests an object from the server.
2. The server responds with the requested object and piggybacks a history in the response header.
3. The client caches the object and stores its history.
4. For problem 1 (caching), when the object is requested, the client determines whether or not to validate the object. The client can choose to use one among multiple policies to reach a decision. For problem 2 (prefetching), the client computes the time of the next refresh using some policy, and refreshes the object at the appropriate time.

The client-side architecture is useful when many individual users share a single cache, e.g., a client-side proxy cache, and can easily support users with diverse preferences with respect to recency of data [6]. This architecture also gives clients greater flexibility to choose how to use histories to estimate updates, and can support adaptive policies (discussed in Section 8) or modeling techniques other than those described in this paper.

5.2 Server-side

In the *server-side* architecture, the server cooperates by computing expiration times of objects in response to a client request. The client does not need to store update history information or perform any additional computation. This architecture is particularly attractive for clients caching data on mobile devices with limited power, bandwidth, and cache space. We note that this architecture could also be implemented using an intermediate proxy to store update histories and perform the computation to reduce loads on both clients and servers.

The *server-side* architecture works as follows:

1. The client requests an object from the server. It can optionally include a parameter, specifying a policy and its desired parameter values in HTTP header.

2. The server computes an expiration time for the expected object using history and includes this expiration time in its response to the client's request.

3. The client caches the object. For solving Problem 1, if the object is requested before the expiration time, it is served from the cache, otherwise it is validated at the remote server. For solving Problem 2, the client refreshes the object at the time computed by the server.

The *server-side* architecture allows clients to benefit from improved accuracy of estimating updates to objects without any added storage or computational overhead. However, since history is not available to the client, the client has less flexibility in determining which policy to use, e.g., it does not have sufficient information to decide on whether a burst occurs.

6 Pull-Based Freshness Policies

We describe three policies for estimating the expected number of updates to objects and scheduling probes to the server, both to sove the caching problem and the prefetching problem. We use the term *refresh* to refer to any probe of a server to check if an object has been updated, and if so, to download the object. A *refresh* can be either a *validate* or a *prefetch*. A *validate* refers to refreshing cached objects in response to a request for the caching problem (Problem 1). A *prefetch* refers to proactively prefetching to refresh cached objects (Problem 2). The computation to estimate the expected number of updates can be performed either at the client-side or the server-side for all three policies and for both problems.

The first policy is based on TTL [10, 22], a widely used heuristic. We develop two new policies, AggHist and IndHist. AggHist and IndHist are inspired by existing work in divergence caching [26] and history-based modeling [20]. All three policies utilize a (different) history model. The specific choice of history model affects the accuracy of estimating the expected number of updates. We develop algorithms for each policy (history model) for both the caching problem and the prefetching problem. We show that the algorithms are optimal with respect to the specific history model (and its estimation). We note however that since the actual pattern of updates may deviate from the model prediction, all these algorithms may not be optimal for real traces and we compare their behavior against three real data traces.

We use $\vec{H}_{ag}$ to denote an aggregate history model (AggHist policy) and $\vec{H}_{ind}$ to denote an individual history model (IndHist policy). $\vec{H}_{ind}$ is derived from updates to a single object. $\vec{H}_{ag}$ is derived from updates to a set of "similar" objects, where similarity depends of the update patterns of the objects. $\vec{H}_{ind}$ may be more accurate in forecasting updates (although there is a danger of over-fitting), yet using it can be expensive. The overhead for the server involves maintaining $\vec{H}_{ind}$ for a potentially large number of objects as well as maintaining indices to rapidly access $\vec{H}_{ind}$ so as not to delay data delivery upon receiving a client request. Also, if all updates to individual objects are sent to a client using a client-side architecture, the amount of storage overhead to any object sent to the client may be excessive. Further, the client needs to analyze the data (online) to determine the next probe. $\vec{H}_{ag}$ is a less costly alternative both with respect to storage overhead and computation by the client.

6.1 Time-to-Live (TTL)

Using only the time an object was last modified, clients or caches can use TTL, a pull-based policy widely used in practice. Recall that we use the generic term TTL to refer to adaptive TTL [22]. TTL estimates how long an object remains fresh as a function of its *last modification time*. Any object that is estimated to be stale must be validated. TTL can be tuned using a parameter α, which is typically a real number between 0 and 1. If an object was last refreshed at the server at time $t_{refresh}$ and was last modified at time $t_{lastmod}$, its TTL is estimated as:

$$TTL = t_{refresh} + \alpha * (t_{refresh} - t_{lastmod}) \tag{6}$$

TTL uses a degenerate form of history, in which the time in between last modification and current time ($t_{refresh} - t_{lastmod}$) serves in estimating the expected time in between updates. Given TTL, the expected interarrival time is computed to be $TTL - t_{lastmod}$. This time serves in designing refresh policies as will be discussed next. For the ensuing discussion, recall that θ is an upper limit on the expected number of updates.

For Problem 1 the TTL policy based probing algorithm is as follows:

Given an object o that was last refreshed at $t_{refresh}$ and requested at time $t_{request}$, and a threshold θ, *use Equation 6 to compute TTL. If* $\frac{t_{request}-t_{lastmod}}{TTL-t_{lastmod}} > \theta$, *i.e., the TTL has expired, validate the object.*

For Problem 2 the TTL policy based probing algorithm works as follows:

Given an object o refreshed at time $t_{refresh}$ and the time $t_{lastmod}$ of the last update before $t_{refresh}$, *calculate* $t_{next\_refresh} = t_{request} + \theta(TTL - t_{lastmod})$, *and refresh o at* $t_{next\_refresh}$.

The optimality of these policies is ensured by the use of θ as an upper bound. The first policy will not probe the server as long as the estimated number of updates is less then θ. The second policy will probe the server as soon as the estimated number of updates is θ.

Tuning the α parameter is left to the designer discretion. Common practice utilizes small α values. Smaller values of α generate more conservative TTL estimates, which improve data freshness, but increase the number of validations. To understand better the reason for this gap, recall that TTL is modeled as a history-based policy, where the interarrival time between updates is computed as $TTL - t_{lastmod}$, using a **single** observation. Clearly, such an estimation is extremely inaccurate. To compensate for the high variability of estimation, more conservative profiles are sought (using $\alpha < 1$). Our hypothesis, which was validated in our experiments as well, is that by using a more conservative approach, many refresh activities are performed in vain. This is because the variance of the estimation is symmetric, which means that many interarrival times are actually higher than $TTL - t_{lastmod}$.

6.2 Aggregate History Based Policy (AggHist)

The aggregate history based policy (AggHist) uses aggregate history ($\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$) of a set of objects. To estimate the update pattern of an individual object from the aggregate update history, servers scale the aggregate λ values by the relative fraction of updates f_o (out of all updates to O) that occurred to o.

For Problem 1, the AggHist policy works as follows:

Given an object o that was last refreshed at $t_{refresh}$ and requested at time $t_{request}$, the aggregated history $\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$, the fraction of updates of $o \in O$ with respect to the total number of updates to O, and a threshold θ, *use Equation 4 to compute the expected number of updates to the object since* $t_{refresh}$. *If the expected number of updates exceeds* θ, *validate the object.*

For Problem 2, the AggHist policy works as follows:

Given an object o refreshed at time $t_{refresh}$, the aggregated history $\vec{H}_{ag} = (\vec{T}, \vec{\lambda})$, the fraction of updates of $o \in O$ with respect to the total number of updates to O, and a threshold θ, *use Equation 4 to calculate the time* $t_{next\_refresh}$ *when the expected number of updates will exceed* θ, *and refresh the object at time* $t_{next\_refresh}$.

Similarly to the TTL policies, the optimality of AggHist policies is ensured by the use of θ as an upper bound. The first policy will not probe the server as long as the estimated number of updates is less then θ. The second policy will probe the server as soon as the estimated number of updates is θ.

As an example, we illustrate the policy for solving Problem 1 on the World Cup trace. Table 2 provides aggregated history of all cyclic objects. Suppose an object $o \in O$ is cached at 1:00 and requested at 8:00. We use the λ values from Table 2. If 1% of all updates at the World Cup site occur to object O, i.e., f_o= 0.01, the corresponding λ values for the intervals from 1:00 to 8:00 in Table 2 are scaled as follows:

Time [1:00 - 7:00]: 23.81 * 0.01 = 0.2381

Time [7:00 - 8:00]: 52.07 * 0.01 = 0.5207

The expected number of updates, using Equation 4, is:

λ_1 * 6 hours + λ_1 * 1 hour = (0.2381 * 6 hours) + (0.5207 * 1 hour) = 1.95

Therefore, if for example the threshold $\theta = 1$, the object will be refreshed.

6.3 Individual History Based Policy (IndHist)

The individual history policy (IndHist) uses the individual history ($\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$) to estimate the freshness of an object.

For Problem 1, the IndHist policy works as follows:

Given an object o that was last refreshed at $t_{refresh}$ and requested at time $t_{request}$, the individual history $\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$, and a threshold θ, *use Equation 4 to compute the expected number of updates to the object since $t_{refresh}$. If the expected number of updates exceeds θ, validate the object.*

For Problem 2, the IndHist policy works as follows:

Given an object o refreshed at time $t_{refresh}$, the individual history $\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$, and a threshold θ, *use Equation 4 to calculate the time $t_{next\_refresh}$ when the expected number of updates will exceed θ, and refresh the object at time $t_{next\_refresh}$.*

Using the same arguments as given for TTL and AggHist, IndHist policies are optimal. As an example to the useage of IndHist policies, consider updates to a single object in the World Cup trace over the 8 day period from June 10-June 17. During this 8 day period, the object had 1 update in the time period [11:00, 12:00) (λ=0.125), 1 update in the time period [12:00, 13:00)(λ=0.125), and 3 updates in [13:00, 14:00) (λ=0.375). If the object was cached at 11:30 and is requested at 14:00, its expected number of updates (using *Equation 4)* is $\frac{1}{2} * 0.125 + 0.125 + 0.375 = 0.5625$.

6.4 Discussion

We briefly discuss the complexity of each of the above policies. TTL can be computed in constant time. Assuming that the histories ($\vec{H} = (\vec{T}, \vec{\lambda})$) are pre-computed for both IndHist and AggHist, the complexity of both policies is linear in the number of intervals in the history. We note that servers may also implement IndHist without pre-computing intervals, i.e., by providing a list of the times the object was updated. This approach reduces computational overhead at servers and gives clients greater flexibility. In this case, the complexity of IndHist for clients is linear in the number of updates in the history because clients must first compute ($\vec{H}_{ind} = (\vec{T}, \vec{\lambda})$), then estimate the number of updates.

In most cases we believe the added computational overhead of AggHist or IndHist will be negligible compared to TTL. Also, AggHist and IndHist are computed only periodically and this cost can be amortized over all client requests. In terms of storage costs, AggHist has a fixed number of intervals, so its size is constant. If storage space or computational complexity of IndHist is an issue, individual histories can be truncated or AggHist can be used.

Choosing the appropriate parameters for each of the above policies depends on update patterns at the data source, client freshness constraints, and resource constraints. Using client-side architecture (see Section 5), the client can use history information to estimate the performance of different policies and parameter settings (as we do in our trace-based experiments in Section 7). Clients can also adjust parameters on the fly, whenever data is insufficiently fresh or excessive resources are consumed. We revisit this issue in Section 7.3.2 using examples from our data traces.

7 Comparison of TTL, IndHist, and AggHist

We now evaluate the AggHist, IndHist, and TTL policies on data traces that exhibit both cyclic and bursty behavior. We show that our proposed history-based policies, AggHist and IndHist, are more effective than the widely-used TTL at meeting our goal of reducing the number of contacts to remote servers while meeting client profiles. We use trace data from two different applications, web caching and email to evaluate these policies. The web caching trace is used to evaluate TTL, AggHist, and IndHist for solving Problem 1, i.e., determining when to refresh cached objects in response to a client's request. The email traces are used to evaluate TTL, AggHist and IndHist for solving Problem 2, i.e., prefetching cached objects to maintain cache freshness.

We first compare TTL and IndHist on the two email traces, which exhibited cyclic update patterns. We then compare TTL, AggHist, and IndHist on both cyclic and bursty objects in the World Cup trace, and motivate the need for a new class of adaptive policies, to be presented in Section 8, that can choose among policies according to an object's update patterns.

7.1 Data Traces

This section provides details of the two data sets we have used in our experiments, namely World Cup data and email data.

7.1.1 World Cup Data

The trace data from the 1998 World Cup web site [3] contains a log of all requests to the site. The World Cup site had servers in four different geographical locations: Paris, France; Herndon, VA; Santa Clara, CA; and Plano, TX. The entire trace consists of 1.3 billion requests made from May 1, 1998 to July 23, 1998. In our experiments we used a 15-day subset of this trace from June 10, 1998 to June 25, 1998. This corresponds to the first 15 days of the World Cup event and includes about 570 million requests. In our experiments, we report separate results for cyclic and bursty objects. It is worth noting that our policies do not require this classification for performing well, as will be discussed in Section 8.2. The separation allows us to analyze performance according to the pattern of updates. To identify objects in each category, we classified objects offline using the update histories from all 15 days of the trace, using the techniques to be described in Section 8.2.

For each request, the trace contains the following:

- *ClientID:* Unique ID of the client making the request. Note that this may be a proxy.
- *ObjectID:* Unique ID of the requested object.
- *Timestamp:* The time the request was made.
- *Size:* Size of the object in bytes.

The trace does not explicitly give information on updates to objects, however, we can infer updates when an object changes size. Many apparent changes in an object's size, however, were caused by temporary inconsistencies at servers in different geographic locations. Our solution to this problem was to only consider an object changed when the majority of requests to the object had the new size, and when the object had this size for at least two minutes. This allowed enough time for updates to propagate to servers in all four locations, to eliminate the effects of false changes due to server inconsistencies. Asa side note, it is worth noting that such a method may fail to fully identify updates. Nevertheless, it serves as a reasonable approximation, which is sufficient for our needs.

In the 15-day trace, 56 million requests were for cyclic objects and 18 million requests were for bursty objects. The remaining 496 million requests were for objects that did not change during the 15 days, most of which were static images. This last category is modeled as cyclic objects with $\lambda = 0$ for all t. This type of object requires no download during the experiment lifetime and was useless for our experiments.

7.1.2 Email Data

Our first email trace (**DBWORLD**) includes email notifications of postings to the DBWORLD electronic bulletin board. The data were collected over seven months and consists of more than 6,400 insertions, from November 9, 2000 through June 17, 2001. Our second email trace (**INBOX**) is taken from messages to a client's inbox from March 3, 2001 - May 24, 2002 and consists of about 10,000 insertions. We collected the data for both these traces using a capture program (similar to the way the vacation program works on Unix) to capture messages and process them.

7.2 Setup

7.2.1 World Cup Experiments

Our experiments with the World Cup trace follow a model of a traditional web caching scenario. When a client requests a cached object, the cache uses the policy to determine whether or not to validate the object. Using TTL, an object is validated if it is requested after it expires. Using IndHist and AggHist, it is validated if the expected number of updates exceeds a specified threshold θ.

We maintained separate caches for each client ID, which may correspond to either an individual client or a proxy. For each client ID, we assumed an initially empty cache. To simplify our presentation, we assume all clients had sufficient space to cache their objects and no objects were evicted from client caches during the trace period. This is a reasonable model because cache size affects only the hit rate of the cache. Therefore, a limited cache would have equal impact on the performance of all estimation policies, and would not change their relative accuracy. Each experiment included a training period to gather object update history information, followed by a test period during which we collected data. We give the length of the training and test periods when reporting the results of each experiment.

Recall that the World Cup trace contains both cyclic and bursty objects. Most bursty objects in the World Cup trace had a "burst" of updates on a single day, and few (if any) updates on other days, as shown in Figure 3. For these objects (or any object with no history available), TTL is likely to provide more accurate freshness estimates compared to the IndHist based policy. A more interesting case occurs when an object that normally has cyclic update patterns experiences a burst in updates. This could occur for example at a news web site that is normally updated at regular intervals but experiences a burst of updates during a breaking news event. For these objects, IndHist is likely to do well during cyclic periods, but TTL may do better during a burst.

We modified the trace data as follows to generate such objects. We randomly selected 55 of the most popular bursty objects with respect to client requests and mapped them to 55 of the most popular cyclic objects (as shown in Figure 4). In our experiments on bursty World Cup objects, we treated each bursty/cyclic pair as a single object. These 55 merged objects exhibited cyclic update patterns for most of the 8 days, but experienced bursts of updates on one day. These were used for the experiments in sections 7.3.2 and 8.2.

7.2.2 Email Experiments

Our experiments with the email traces model a scenario where a client has a locally cached mailbox, e.g., on their mobile device, that needs to be refreshed in the background to promptly notify the client of new messages. This scenario also applies to source monitoring applications where clients need to receive updates in a timely manner [38], and other instances of the prefetching problem from Section 3.2.

For the email application, we compare the TTL and IndHist policies. After each refresh, for the TTL policy, we computed the time of the next refresh as a function of the time the last message arrived. For the IndHist policy, after each refresh we computed the time of the next refresh as the time that the expected number of updates (i.e., new messages) would exceed some threshold θ. We used the first week of each trace as a training period to gather a history.

7.2.3 Metrics

We use the following metrics:

- **Total Validations**: This is the number of times requested objects that were in the cache needed to be validated at the remote server.
- **Useful Validations**: This is the number of validated objects that had actually been modified at the remote server.
- **Freshness Misses**: This is the number of validated objects that had not been modified at the remote server. These validations add unnecessary overhead and consume bandwidth without improving data freshness.
- **Stale Hits**: For the World Cup trace, this is the number of objects that were served from the cache without validation but had actually been updated at the remote server.
- **Average Delay**: For the email traces, this is the average amount of time elapsed between the arrival of a new message and the time it appears in the client's mailbox.

By changing α, θ, and other threshold numbers, we tested each policy. This step returned a varying number of validations and average delay (for email traces) or stale hits (for the World Cup trace). We then plotted all the results for each policy for the same validation numbers in a single graph, producing the graphs in the paper.

7.3 Results

Our experiments show that using either IndHist or AggHist for cyclic objects can *significantly improve the accuracy of estimates of an object's freshness.* In web caching, improving the accuracy of estimates can increase the number of objects served from the cache without validation, which in turn reduces costly remote server accesses for clients and reduces unnecessary contacts with servers. (Recall that freshness misses may be as many as 30-50% of all cache hits [15]). In email applications, improving the accuracy of estimates can reduce the delay of new messages appearing in a client's mailbox without increasing the mailbox refresh rate, which is of particular importance to mobile devices. Accurately estimating when updates occur enables capturing important updates in a timely manner.

7.3.1 Accuracy of Estimates

World Cup Trace We first compare the accuracy of estimating the number of updates to cyclic objects in the World Cup Trace using TTL, IndHist, and AggHist. Each time a client requests a cached object, we compare the actual number of updates to the object against the estimated number using each policy. Using TTL, we estimate the number of updates to an object at time t as $(t -$

$t_{lastmod})/(TTL - t_{lastmod})$, where $t_{lastmod}$ is the last modified time of the object, and use an α value of 0.05 (see Section 6.1), which is typical of values used in practice, e.g, in the Squid cache [7]. We note that other values of α do not show significantly different results, and we provide an explanation of why the α value has little effect below. For IndHist and AggHist, we calculate the estimated number of updates to an object as described in Section 6.

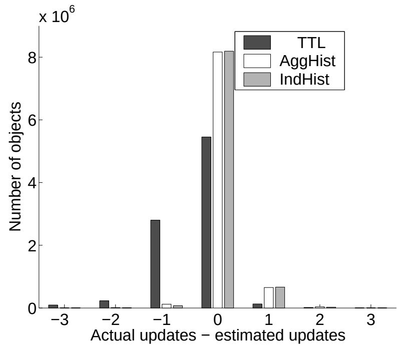
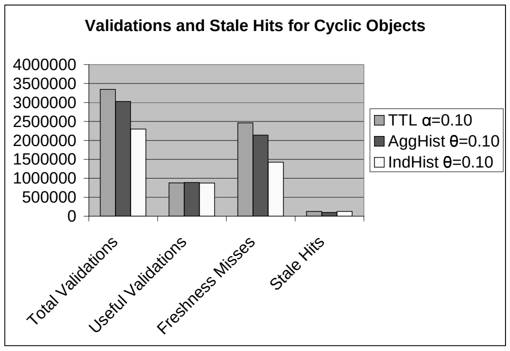


Figure 5: Comparison of three policies for Cyclic objects in the World Cup Trace

Figure 5 compares the estimated updates to the actual value for each policy. A value of 0 means the estimate was *accurate*. A positive error value means the actual value *exceeded* the estimated value, and a negative value means the actual value was *less* than estimated. AggHist and IndHist have nearly *twice* as many accurate estimates as TTL. This shows that using histories can significantly improve the accuracy of freshness estimates for cyclic objects.

We now explain why changing the value of α has little effect on the accuracy of TTL relative to IndHist and AggHist. As observed in Figure 5, TTL frequently overestimates the number of updates because it (sometimes incorrectly) estimates that a recently updated object is likely to be updated in the near future. However, a recently updated cyclic object may not be updated again until the next cycle (e.g., the next day). Thus, for a recently updated cyclic object, even high α values will not significantly affect the accuracy of the estimated number of updates.

We now compare the TTL, IndHist, and AggHist policies in terms of both bandwidth consumption and data freshness for cyclic objects. We show the results for selected α and θ values in Figure 6. The first observation is that AggHist and IndHist perform significantly fewer validations than TTL, without a significant increase in the number of stale hits. When $\theta = 0.20$, AggHist and IndHist have both fewer validations *and* fewer stale hits than TTL α =0.20. The θ values can be tuned according to the latency and recency requirements of an application [6]. As we will show in Section 7.3.2, both IndHist and AggHist can provide comparable recency to TTL with lower bandwidth consumption,



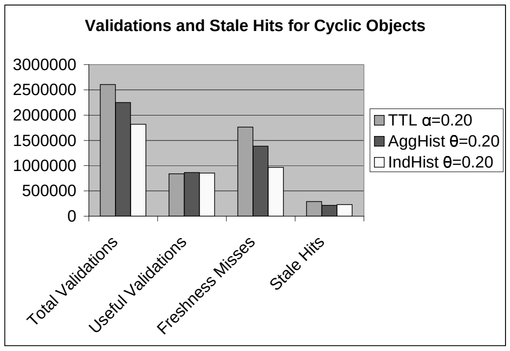


Figure 6: Comparison of three staleness estimation policies for Cyclic objects

for a wide range of α and θ values. Thus, policies that consider past updates to objects can significantly reduce bandwidth consumption compared to TTL while providing fresher data to clients.

Email Traces We next consider the accuracy of the IndHist policy for the email application. We do not consider AggHist for this application because the trace consists of a single object (the mailbox). Recall that for the email application, using IndHist we refreshed the mailbox whenever the expected number of updates (new messages) exceeded θ. In Figure 7 we compare the expected number of these updates per validation against the actual average number of updates per validation. Intuitively, if the average number of updates per validation is close to the expected number of updates θ, then the model provides a good estimate. The solid line in Figure 7 indicates the case when the actual number of updates equals θ. For the DBWORLD trace, the estimated number of updates is very close to θ. While the estimates are slightly less accurate for the INBOX trace, they are still close to the solid line for all values of θ, which shows that IndHist accurately estimates the number of updates.

7.3.2 Number of Validations

We now report on the number of validations required to maintain a given level of freshness.

Email Traces We first consider the DBWORLD and INBOX traces. Recall that we use the average delay as our metric. We tune TTL by varying α between 0 and 1 and IndHist by varying θ between 0 and 1. We plot the number of validations against the average delay for TTL (for different α values) and IndHist

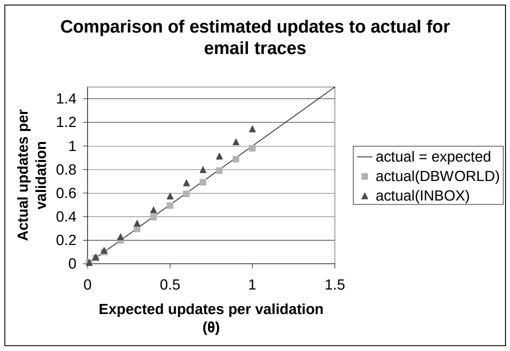


Figure 7: Comparison of the expected and actual updates per validation using IndHist

(for different θ values) in Figure 8. As expected, as the number of validations increases, the average delay decreases. The key observation is that for any given average delay, IndHist performs significantly fewer validations than TTL. For example, in Figure 8(a), to provide an average delay of about 300 seconds, TTL must perform about 170,000 refreshes while IndHist performs about 80,000, i.e., a 47% reduction in the number of validations. Similarly, in Figure 8(b), to provide an average delay of 500 seconds TTL performs about 50,000 validations while IndHist performs about 20,000, i.e., a 60% reduction. Thus, IndHist can reduce the total number of refreshes by more than half. This can provide significant savings in terms of both power and bandwidth to clients who read email on their mobile devices. Recall that the email traces generally exhibited cyclic behavior. These results show that IndHist can indeed perform better than TTL for cyclic objects.

World Cup Trace Next, we compare the TTL, IndHist, and AggHist in terms of both number of validations and data freshness of both cyclic and bursty objects in the World Cup trace. In the experiments for cyclic objects, we used all 15 days of trace data. We used the first 8 days to construct histories, and ran the experiments on the next 7 days. In the experiments for bursty objects, we used 8 days of trace data and performed preprocessing as described in Section 8.2. We used the first 4 days to construct individual update histories, and ran the experiments on the last 4 days. For the TTL policy, we varied α from 0.05 to 0.7, and for the IndHist and AggHist policies we varied θ from 0.05 to 0.7. We note that α and θ are unrelated parameters used for different policies. However, recall that our purpose is to compare the number of stale hits of each policy for a given number of validations. For TTL, varying α varies the

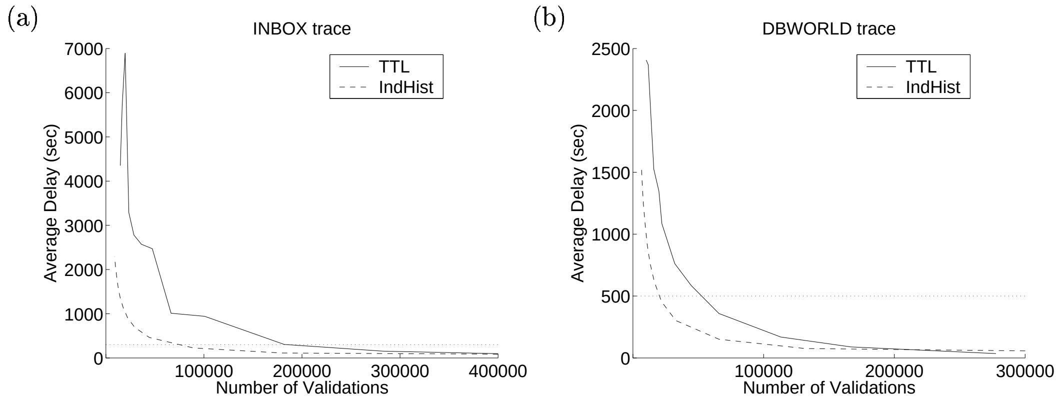


Figure 8: Effect of Tuning TTL and IndHist on average delay and validations

number of validations, and similarly varying θ varies the number of validations for AggHist and IndHist. Further, note that for the same θ value different policies may perform a different number of validations (as shown below in Table 3).

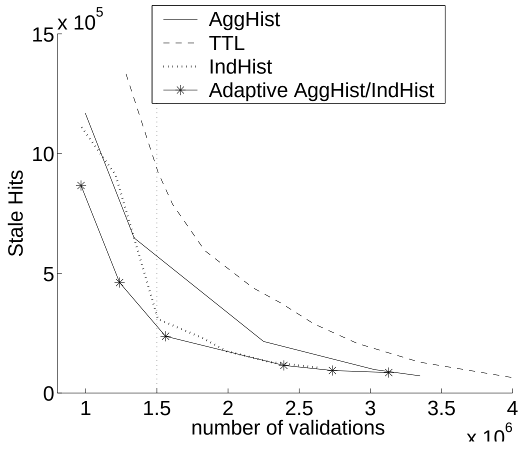


Figure 9: Effect of Tuning TTL, AggHist, and IndHist on data freshness and validations for cyclic World Cup objects

In Figure 9 we report on the number of stale hits given similar levels of total validations. Note that for all values of θ, IndHist did not go beyond 2,500,000 validations and AggHist did not go beyond 3,500,000 validations. For all three policies, increasing the total number of validations reduces the number of stale hits. Given the same number of validations, both AggHist and IndHist deliver significantly fewer stale objects than TTL. This is because the improved accuracy of the freshness estimates of objects reduces the number of unnecessary validations, and shows once again that using histories can perform better than TTL for cyclic objects. This is especially true when there are relatively few validations, i.e., higher values of α and θ. For example, when each of the policies has about 1,500,000 total validations, TTL ($\alpha \approx 0.5$) provides $\approx$800,000 stale hits while AggHist ($\theta \approx 0.5$) provides $\approx$500,000 stale hits and IndHist ($\theta \approx 0.3$) provides $\approx$300,000 stale hits.

Validations	TTL	AggHist	IndHist	Adaptive
$\approx$ 1,000,000	$\alpha > 0.7$	θ=0.7	θ=0.5	α=θ=0.7
$\approx$ 1,500,000	α=0.5	θ=0.5	θ=0.3	α=θ=0.3
$\approx$ 2,000,000	α=0.3	θ=0.2	θ=0.15	α=θ=0.1

Table 3: α and θ values corresponding to a given number of validations for each policy

In Table 3 we show the α and θ values that correspond to an approximate number of validations, for each of the policies shown in Figure 9. We note that for the parameter settings of each of the algorithms, the number of validations is approximate. The key observation in this table is that different α or θ settings for two different policies may result in a comparable number of validations. Thus, the choice of parameter settings for a given policy should be based on the expected number of validations or data freshness, as discussed below.

Clients can choose a θ value according to either their desired freshness of the data, or desired number of validations, and adjust the θ value if current settings do not meet their freshness or resource constraints. If sufficient history is available, clients can also leverage knowledge of past performance to choose appropriate parameter settings. In the above example, a client using AggHist who wants a low number of validations might initially choose θ=0.7. Similarly, a client using IndHist who wants no more than 300,000 stale hits could choose θ=0.3.

A key observation is that IndHist offers an overall improvement over AggHist because it can model the individual update patterns of objects that may differ from the average behavior. However, individual history is not always a good predictor of updates for cyclic objects. If there is insufficient history information available, the IndHist policy may not be able to accurately predict when updates will occur. In contrast, since AggHist captures the behavior of objects with similar update patterns, it is better suited to deal with new objects whose history is too short to yet be stable. Figure 9 compares an adaptive IndHist/AggHist policy (described in detail in Section 8) that can combine the benefits of both policies. This adaptive policy *performs better than either policy alone.* (We note that in Figure 9 the T_{ind} parameter described in Section 8 is set to 0.5.) The adaptive IndHist/AggHist policy chooses between IndHist and AggHist based on the available history information, as described in the next section.

Figure 10 compares TTL and IndHist on the bursty objects in the World

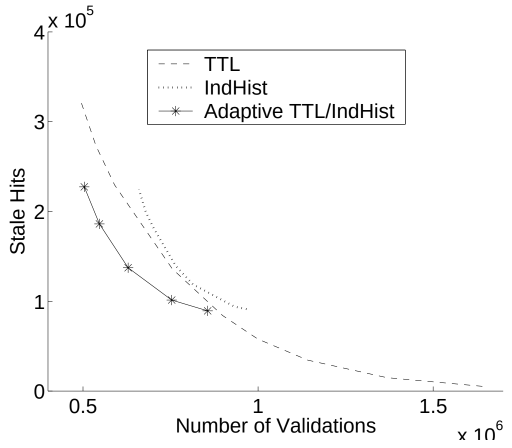


Figure 10: Effect of Tuning TTL and IndHist on data freshness and validations for bursty World Cup objects

Cup trace. AggHist performed significantly worse than both of these and we do not show these results. As expected, TTL performs better than IndHist (yet not significantly better) because it can more accurately predict when updates occur during bursty periods. However, we expect that IndHist is more effective in non-bursty periods. To illustrate, an adaptive TTL/IndHist policy is shown in Figure 10; it can perform better than TTL alone because it uses IndHist to estimate updates during non-bursty periods. In Section 8 we evaluate this adaptive TTL/IndHist policy on both cyclic and bursty objects.

8 Adaptive Pull-Based Policies

Figures 9 and 10 illustrate that adaptive policies that can choose among multiple policies outperform individual policies. There are several challenges in developing an effective adaptive policy. One is selecting a pool of individual policies. The second is determining the criteria to choose (alternate) among the policies, and the third is making sure the choice is beneficial. We present two adaptive policies, Adaptive IndHist/AggHist and Adaptive IndHist/TTL. We note that there are many other adaptive policies that could be used, and we do not claim that these two are the best. Rather, our contribution is in motivating the use of adaptive policies, and illustrating their use and benefits.

8.1 Adaptive IndHist/AggHist

We have observed that for cyclic objects, IndHist works well when there is sufficient individual history available to predict when the object is likely to be updated. In contrast, AggHist works well for objects with cyclic behavior, but whose histories are too short to be stable. An adaptive policy can exploit both the aggregate and individual behavior for cyclic objects.

We formalize this policy below. Recall that symbols used in this paper are defined in Table 1.

8.1.1 Policy

Let *NumUpdates* be the size of the instantiation set used to estimate a history and suppose that the number of intervals in $\vec{H}_{ind}$ is K, representing the different intensities of our RPC model. We denote by *NumSub* the number of intervals in $\vec{H}_{ind}$ for which an update was recorded. The Adaptive IndHist/AggHist policy is as follows: *Given an individual history and a parameter* T_{ind}*, we compute the ratio* NumSub/NumUpdates*. If* NumSub/NumUpdates $> T_{ind}$*, AggHist is used; else IndHist is used.*

The lower the threshold T_{ind} is, the more updates are required to switch to IndHist. T_{ind} serves as a measure of confidence in the individual history. If the total number of updates to the object *NumUpdates* is equal to *NumSub*, *i.e.*, the ratio $NumSub/NumUpdates = 1$, then the object was updated once in each interval of $\vec{H}_{ind}$, suggesting insufficient history to accurately model updates using IndHist. Therefore, T_{ind} should be typically restricted to be less than 1. We note that the adaptive policy is based on the ratio of *NumUpdates* and *NumSub* and is independent of the value of K. (Recall that we discussed choosing an appropriate K value in Section 4.2.)

Intuitively, the ratio gives us a confidence measure of how likely future updates are to occur in the same intervals as previously observed updates. For example, if an object has 10 updates and all occurred between 1:00 and 2:00, (*i.e.*, $NumSub/NumUpdates = 0.1$), we hypothesize that future updates are likely to occur in the same interval, so IndHist would be a good predictor. On the other hand, if the object has 10 updates, each of which occurred in a different interval and only one of which occurred between 1:00-2:00 (*i.e.*, $NumSub/NumUpdates = 1$), we have considerably less confidence that another update will occur between 1:00 and 2:00.

Note that for T_{ind}=1, IndHist is always selected, and for T_{ind}=0 AggHist is selected. Clients can more aggressively choose AggHist or IndHist by varying the value of T_{ind}.

8.1.2 Results

We report on Adaptive IndHist/AggHist for different T_{ind} values for cyclic objects in the World Cup Trace.

Figure 11 plots the number of validations compared to the number of stale hits. We present curves for three different values of T_{ind}, as well as for TTL, IndHist, AggHist, and IndHist- 10 day (described below). We first consider the impact of using a shorter update history or limited update information on the accuracy of the IndHist policy. We limited the length of the history to 10 days prior to the time the object was cached (labeled IndHist- 10 day in Figure 11).

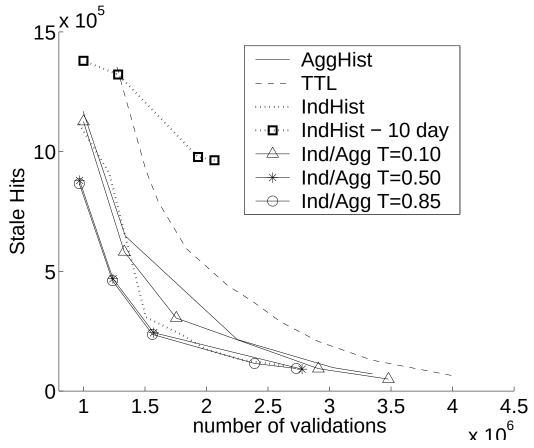


Figure 11: Comparison of Adaptive IndHist/AggHist to TTL, IndHist, and AggHist

Note that in contrast, the policy labeled IndHist includes all prior updates to an object in the trace, up to all 15 days. For many objects, the IndHist-10 day policy does *worse* than TTL and AggHist. This motivates the need for sufficient history for IndHist, and shows the benefits of AggHist when there is insufficient history available for an individual object.

Next, we compare the performance of the Adaptive IndHist/AggHist policy to IndHist (with complete update histories) and AggHist alone. Adaptive IndHist/AggHist with T_{ind} ranging from 0.50 to 0.85 outperforms TTL, IndHist, and AggHist. When T_{ind}= 0.1, the policy performs closer to AggHist as expected. These results suggest that Adaptive IndHist/AggHist is not very sensitive to the selection of T_{ind} and any value around 0.5 that allows the adaptive policy to switch is effective.

8.2 Adaptive IndHist/TTL

For bursty objects, the IndHist policy performs best during the non-bursty periods, but performs poorly when objects experience bursts. Further, when it uses a full update history, including bursty periods, it may estimate updates less accurately after the bursty period. In contrast, TTL performs best during bursty periods because it assumes that objects that were recently updated are likely to be updated again soon. An adaptive IndHist/TTL policy combines the best features of both policies, as illustrated in Figure 10.

Our adaptive IndHist/TTL policy can detect bursts and dynamically chooses between IndHist and TTL. Thus, it can generalize well to different types of update patterns, and requires no prior knowledge of whether an object is cyclic or bursty. We first describe how we detect bursts. We then describe the adaptive policy (Adaptive IndHist/TTL) which dynamically chooses between the IndHist and TTL policies depending on whether or not an object exhibits bursty behavior. Thus, for objects with no bursts, it has comparable performance to IndHist.

8.2.1 Burst Detection and Adaptive IndHist/TTL policy

In Section 4.3 we have defined a burst as a deviation of an instantiation from the model. In particular, we are interested in bursts in which the update intensity is increased. Such deviation can be detected whenever the number of actual updates to an object is considerably higher than that approximated by IndHist. Consider an object that is cached at time t and created at time t_0. We estimate that a burst occurs when the *actual number of updates* in a window of size W prior to t, i.e., all updates in the interval $[t-W,\, t]$, exceeds the *expected number of updates.* The *expected number of updates* is estimated by IndHist, using only updates that occurred in $[t_0,\, t-W]$ prior to the current cycle.

The adaptive history policy works as follows: *Given an intensity function λ for the interval (t_0, t - W), and λ^* for the interval ($t-W$, t), a distance function $f(\lambda, \lambda^*)$, and a threshold T_{burst}, Adaptive IndHist/TTL identifies a burst if $f(\lambda, \lambda^*) \geq T_{burst}$. On each request, if $f(\lambda, \lambda^*) \geq T_{burst}$, Adaptive IndHist/TTL assumes a burst is occuring and uses TTL. Else, if $f(\lambda, \lambda^*) < T_{burst}$, Adaptive IndHist/TTL assumes a cyclic behavior and uses the IndHist policy.*

We next provide a distance measure f. This was empirically evaluated to provide a good estimation of bursty periods in the World Cup trace data. We note that more research is needed to identify distance measures that will work on several traces. Let the *expected number of updates* (Λ) in $[t-W,\, t]$ with respect to time t be $\Lambda(W,t)$, and the *actual number of updates* (Λ^*) in $[t-W,\, t]$ be $\Lambda^*(W,t)$. Then,

$$f(\lambda, \lambda^*) = \begin{cases} \frac{\Lambda^*(W,t)}{\Lambda(W,t)} & \text{if } \Lambda(W,t) > 0 \quad (a) \\ T_{burst} & \text{if } \Lambda(W,t) = 0 \text{ and } \Lambda^*(W,t) > 0 \quad (b) \\ 0 & \text{otherwise} \quad (c) \end{cases}$$

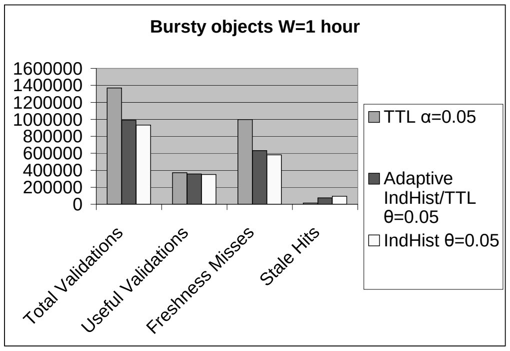
Intuitively, condition (a) covers the case when at least one update was expected ($\Lambda(W,t) > 0$). A burst occurs when the ratio of *observed* updates to *expected* updates exceeds T_{burst}. Condition (b) covers the case when no updates were expected ($\Lambda(W,t) = 0$) and at least one update occurs. Note that when T_{burst}=0, this policy is identical to TTL, and when T_{burst}=∞, this policy is identical to IndHist, thus, it is a generalization of both policies.

8.2.2 Results

We compare TTL, IndHist, and Adaptive IndHist/TTL. We evaluate the policies on both the "combined" trace of 55 merged objects and on the remaining cyclic objects. We ran these experiments on the first 8 days of our 15 days of trace data. We used the first 4 days to gather history information, and report results on the remaining 4 days. For comparison purposes, we also report on results for the cyclic objects during the same period.

For Adaptive IndHist/TTL, recall that we estimate when a burst occurred by considering the number of updates in a window W. Adaptive IndHist/TTL will use TTL whenever $f(\lambda, \lambda^*)$ in a window of size W exceeds the threshold T_{burst}. In our experiments, we report results for $W = 1$ hour and $W = 24$ hours, and $T_{burst} = 2$.

(a)



(b)

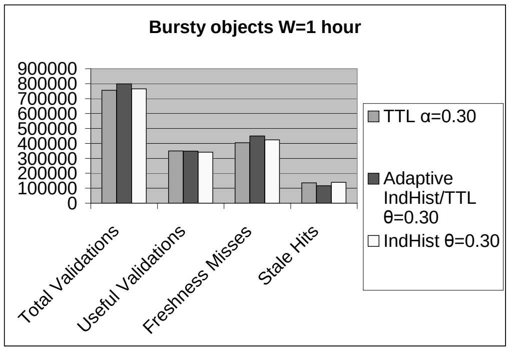


Figure 12: Validations and Stale Hits for Bursty objects when W=1 hour

For the TTL policy, we varied α from 0.02 to 0.7, for IndHist we varied θ from 0.02 to 0.7, and for the Adaptive Policy we set α=θ and varied this value from 0.02 to 0.7. We show the number of validations and stale hits for selected values of α (TTL) and θ (IndHist) when W= 1 in Figure 12. In Figure 12 (a), the number of useful validations are similar for each of the policies, while TTL has about 40% more validations. Adaptive IndHist/AggHist has fewer stale hits than IndHist. In Figure 12 (b), the total number of validations of all three techniques are similar. We note that Adaptive IndHist/AggHist has fewer stale hits than TTL. It also has fewer stale hits than IndHist, and is thus better suited to bursts.

We plot the number of stale hits versus the total number of validations in Figure 13(a). As expected, TTL outperforms IndHist for the bursty objects. This is because TTL assumes that objects that have been updated recently are more likely to be updated soon, so it is well-suited during bursty periods. In contrast, IndHist assumes that an object's update patterns will be consistent with its past update history, so it cannot handle bursts as well. However, Adaptive IndHist/TTL offers improvement over IndHist, especially as the total number of validations increases. This shows that Adaptive IndHist/TTL can detect some bursts in updates and chooses TTL when appropriate. Adaptive IndHist/TTL with W=24 provides fewer stale hits, and in most cases provides

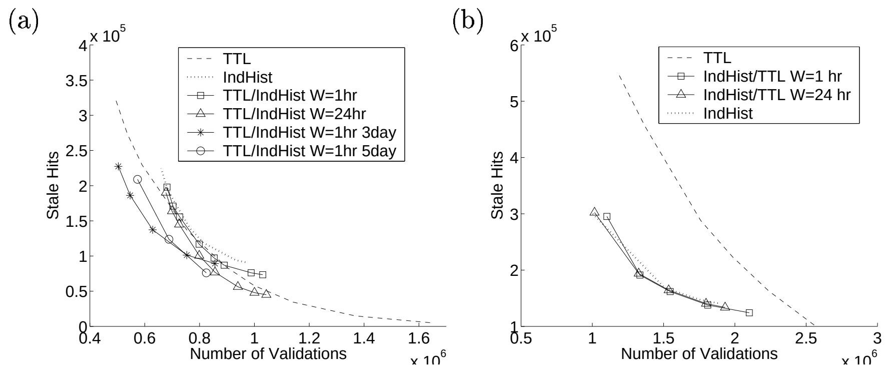


Figure 13: Effect of Tuning TTL, IndHist, and Adaptive IndHist/TTL on data freshness for (a) bursty and (b) cyclic objects

fresher data than TTL for the same number of validations. This suggests that larger values of W may be more effective at detecting bursts.

We also consider the effects of truncating update histories for bursty objects. We consider maintaining a sliding window of individual update histories and ignoring updates that occured outside this window. We consider windows of both 3 days and 5 days prior to the time an object was cached. We show these results in Figure 13 (a) for Adaptive IndHist/TTL with W=1 . The interesting observation is that using shorter update histories makes the adaptive policy perform better than TTL. This contrasts with the results in Figure 11, which showed that shorter update histories caused IndHist to perform worse than TTL for cyclic objects.

We hypothesize that shorter update histories perform well for objects that experience bursts because they allow ignoring the bursty period. This improves the accuracy of the IndHist policy. In contrast, shorter update histories make IndHist perform worse for cyclic objects because they decrease the amount of available history information.

Thus, our results show that (1) Adaptive IndHist/TTL performs better than either TTL or IndHist alone for objects with both cyclic and bursty periods and (2) when using histories to predict updates, it is important to ignore bursty periods that are not consistent with the rest of the history. Ignoring bursty periods (by using shorter histories) significantly improves the accuracy and effectiveness of Adaptive IndHist/TTL. Developing techniques to detect bursts, and to ignore them for history construction is an area for future work.

We also compare the performance of IndHist and Adaptive IndHist/TTL on the cyclic objects over the same 8-day period. Our goal is to ensure that Adaptive IndHist/TTL performs as well as IndHist on cyclic objects. We plot these results in Figure 13 (b). The key observation is that Adaptive IndHist/TTL has comparable performance to IndHist for cyclic objects, so it can generalize to both cyclic and bursty objects without requiring any a priori classification of an object's behavior.

9 Conclusions

The challenge of efficient data delivery across wide area networks is becoming increasingly important as the number and diversity of wide area applications continues to grow. As push-based data delivery solutions are difficult to widely deploy and may not scale to the growing number of clients, pull-based data delivery is becoming the preferred solution for rapid and widespread deployment of wide area applications.

In this paper we have presented a framework for pull-based wide area data delivery, which scales to a large number of clients and servers. We have presented a model of updates to data sources using object update histories, and a set of policies that use the model to estimate when updates occur to objects. We have presented a set of architectures that enable deployment of our policies with minimal modifications to both clients and servers. These include a server-side architecture that reduces overhead for clients at the cost of flexibility, and a client-side architecture that gives more control to clients and minimizes server overhead. We have also presented a set of adaptive policies to cope with update bursts or to estimate the behavior of objects with insufficient histories available. Our experimental evaluation of our policies using trace data from two very different wide area applications shows that our policies can indeed reduce communication overhead with servers while providing comparable data freshness to existing pull-based policies.

This work serves as a first step towards scalable pull-based data delivery that minimizes communication between clients and servers while meeting client freshness requirements. Future work involves a more thorough analysis of burst detection and the identification of more adaptive policies for managing the uncertainty involved in the use of stochastic update models when estimating update events.

Acknowledgments

We thank Chris Olston for helpful comments on an earlier draft of this paper.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM. TODS Vol. 15, no. 3*, 1990.

[2] K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On Space Management in a Dynamic Edge Data Cache. *Proceedings of WebDB*, 2002.

[3] M. Arlitt and T. Jin. 1998 world cup web site access logs. *Available at http://www.acm.org/sigcomm/ITA/*, 1998.

[4] E. Baralis, S. Paraboschi, and E. Teniente. Materialized view selection in a multidimensional database. *Proc. VLDB*, 1997.

[5] L. Bright, S. Bhattacharjee, and L. Raschid. Supporting Diverse Mobile Applications with Client Profiles. *Proceedings of ACM Workshop on Wireless Mobile Multimedia (WoWMoM)*, 2002.

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

[7] Squid Proxy Cache. *http://www.squid-cache.org.*

[8] K.S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. *Proc. SIGMOD*, 2001.

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

[10] V. Cate. Alex - a global filesystem. *Proc. USENIX File System Workshop*, 1992.

[11] J. Challenger, A. Iyengar, and P. Dantzig. A Scalable System for Consistently Caching Dynamic Web Data. *Proceedings of IEEE INFOCOM*, 1999.

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

[13] J. Cho and H. Garcia-Molina. Estimating frequency of change. *ACM Transactions on Internet Technology (TOIT)*, 3(3):256–290, 2003.

[14] J. Cho and A. Ntoulas. Effective change detection using sampling. *Proc. VLDB Conf.*, 2002.

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

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

[17] J. Eckstein, A. Gal, and S. Reiner. Optimal information monitoring under a politeness constraint. Technical Report RRR 16-2005, RUTCOR, Rutgers University, 640 Bartholomew Road, Busch Campus Piscataway, NJ 08854 USA, May 2005.

[18] M. Franklin, M. Carey, and M. Livny. Transactional Client-Server Cache Consistency: Alternatives and Performance. *ACM Transactions on Database Systems*, 22(3):315–363, 1997.

[19] M. Franklin and S. Zdonik. Data in your face: Push technology in perspective. *Proc. ACM SIGMOD Conference*, 1998.

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

[21] H. Gupta. Selection of views to materialize in a data warehouse. *Proc. ICDT*, 1997.

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

[23] BlackBerry Wireless Handhelds. *http://www.blackberry.com.*

[24] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes efficiently. *Proc. ACM SIGMOD Conference*, 1996.

[25] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, 1988.

[26] Y. Huang, R. Sloan, and O. Wolfson. Divergence caching in client-server architectures. *Proc. PDIS*, 1994.

[27] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. *Proceedings of the 19th International Symposium of Computer Architecture (ISCA)*, 1992.

[28] B. Krishnamurthy and C. Wills. Study of piggyback cache validation for proxy caches in the world wide web. *Proc. USENIX Symposium on Internet Technologies and Systems*, 1997.

[29] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. *Proc. VLDB*, 2001.

[30] T.L. Lai and J.Z. Shan. Efficient recursive algorithms for detection of abrupt changes in signal and control systems. *IEEE Transactions on Automatic Control*, 44:952–964, 1999.

[31] J.-J. Lee, K.-Y. Whang, B. S. Lee, and J.-W. Chang. An update-risk based approach to TTL estimation in web caching. *Proc. Conference on Web Information Systems Engineering (WISE)*, 2002.

[32] C. Liu and P. Cao. Maintaining strong cache consistency on the world wide web. *Proc. ICDCS*, 1997.

[33] R.L. Mason, Y.-M. Chou, J.H. Sullivan, Z. G. Stoumbos, and J. C. Young. Systematic patterns in T^2 charts. *Journal of Quality Technology*, 35:47–58, 2003.

[34] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[35] K. Nishina. A comparison of control charts from the viewpoint of change-point estimation. *Quality and Reliability Engineering International*, 8:537–541, 1992.

[36] C. Olston, B.T. Loo, and J. Widom. Adaptive precision setting for cached approximate values. *Proc. ACM SIGMOD Conference*, 2001.

[37] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. *Proc. ACM SIGMOD Conference*, 2002.

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

[39] S. Pandey, K. Ramamritham, and S. Chakrabarti. Monitoring the dynamic web to respond to continuous queries. *Proceedings of the WWW Conference*, 2003.

[40] M.R. Reynolds, Jr. and Z.G. Stoumbos. A general approach to modeling CUSUM charts for a proportion. *IIE Transactions on Quality and Reliability Engineering*, 32:515–535, 2000.

[41] S. Ross. *Stochastic Processes.* Wiley, second edition, 1995.

[42] Z.G. Stoumbos, M.R. Reynolds Jr., and W.H. Woodall. Control chart schemes for monitoring the mean and variance of processes subject to sustained shifts and drifts. *to appear in the Handbook of Statistics: Statistics in Industry*, 22, 2003. C.R. Rao and R. Khattree (eds.).

[43] H.M. Taylor and S. Karlin. *An Introduction to Stochastic Modeling.* Academic Press, 1994.

[44] E. Yashchin. Estimating the current mean of a process subject to abrupt changes. *Technometrics*, 37:311–323, 1995.

[45] J. Yin, L. Alvisi, M. Dahlin, and A. Iyengar. Engineering Server-Driven Consistency for Large Scale Dynamic Web Services. *Proceedings of 10th WWW Conference*, 2001.

[46] H. Zhu and T. Yang. Class-based Cache Management for Dynamic Web Content. *Proceedings of IEEE INFOCOM*, 2001.