# Dissemination of Dynamic Data on the Internet

*Krithi Ramamritham*
*Pavan Deolasee*
*Amol Katkar*
*Ankur Panchbudhe*
*Prashant Shenoy*

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India 400076
*and*
Department of Computer Science
University of Massachusetts
Amherst MA 01003
Email: krithi@cse.iitb.ernet.in

**Abstract.** Dynamic data is data which varies rapidly and unpredictably. This kind of data is generally used in on-line decision making and hence needs to be delivered to its users comforming to certain time or value based application-specific requirements. The main issue in the dissemination of dynamic web data such as stock prices, sports scores or weather data is the maintenance of *temporal coherency* within the user specified bounds. Since most of the web servers adhere to the HTTP protocol, clients need to frequently *pull* the data depending on the changes in the data and user's coherency requirements. In contrast, servers that possess *push* capability maintain state information pertaining to user's requirements and push only those changes that are of interest to a user. These two canonical techniques have complementary properties. In pure pull approach, the level of temporal coherency maintained is low while in pure push approach it is very high, but this is at the cost of high state space at the server which results in a less resilient and less scalable system. Communication overheads in pull-based schemes are high as compared to push-based schemes, since the number of messages exchanged in the pull approach are higher than in push based approach. Based on these observations, this paper explores different approaches to combining the two approaches so as to harness the benefits of both approaches.

## 1 Dynamic Data Dissemination

Dynamic data can be defined by the way the date changes. First of all it changes *rapidly*, changes can even be of the order of one change every few seconds; it also changes *unpredictably*, making it very hard to use simple prediction techniques or time-series analysis. Few examples of dynamic data are stock quotes, sports scores and traffic or weather data. Such of kind of data is generally used in decision making (for example, stock trading or weather forecasting) and hence the timeliness of delivery of this data to its users becomes very important.

Recent studies have shown that an increasing fraction of the data on the world wide web is dynamic. Web proxy caches that are deployed to improve user response times must track such dynamic data so as to provide users with temporally coherent information. The coherency requirements on a dynamic data item depends on the nature of the item and user tolerances. To illustrate, a user may be willing to receive sports and news information that may be out-of-sync by a few minutes with respect to the server, but may desire stronger coherency requirements on data items such as stock prices. A proxy can exploit user-specified coherency requirements by fetching and disseminating only those changes that are of interest and ignoring intermediate changes. For instance, a user who is interested in changes of more than a dollar for a particular stock price need not be notified of smaller intermediate changes. The problem can be termed as the *problem of maintaining desired temporal coherency* between the source and the user, with the proxy substantially improving the access time, overheads and coherency.

We study mechanisms to obtain timely updates from web sources, based on the dynamics of the data and the users' need for temporal accuracy, by judiciously combining push and pull technologies and by using proxies to disseminate data within acceptable tolerance. Specifically, the proxies (maintained by client organizations) ensure the temporal coherence of data, within the tolerance specified, by tracking the amount of change in the web sources. Based on the changes observed and the tolerance specified by the different clients interested in the data, the proxy determines the time for *pull*ing from the server next, and pushes newly acquired data to the clients according to their temporal coherency requirements.

Of course, if the web sources themselves were aware of the clients' temporal coherency requirements and they were endowed with *push* capability, then we can avoid the need for mechanisms such as the ones proposed here. Unfortunately, this can lead to scalability problems and may also introduce the need to make changes to existing web servers (which do not have push capabilities) or to the HTTP protocol.

## 2 Maintaining Temporal Coherency

Consider a proxy that caches several time-varying data items. To maintain coherency of the cached data, each cached item must be periodically refreshed with the copy at the server. For highly dynamic data it may not be feasible to maintain strong cache consistency. An attempt to maintain strong cache consistency will result in either heavy network overload or server load. We can exploit the fact that the user may not be interested in every change happening at the source to reduce network utilization as well as server overload.

We assume that a user specifies a temporal coherency requirement $c$ for each cached item of interest. The value of $c$ denotes the maximum permissible deviation of the cached value from the value at the server and thus constitutes the user-specified tolerance. Observe that $c$ can be specified in units of *time* (e.g., the item should never be out-of-sync by more than 5 minutes) or *value* (e.g., the stock price should never be out-of-sync by more than a dollar). As shown in figure 1, the proxy sits between the user and the server, and handles all communication with the server based on the user constraint. Given the value of $c$, the proxy can use push- or pull-based techniques to ensure that that the temporal coherency requirement ($tcr$) is satisfied.

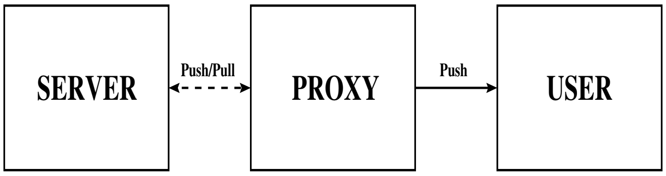


**Fig. 1.** Proxy-based Model

The *fidelity* of the data seen by users depends on the degree to which their coherency needs are met. We define the fidelity $f$ observed by a user to be the total length of time that the above inequality holds (normalized by the total length of the observations). In addition to specifying the coherency requirement $c$, users can also specify their fidelity requirement $f$ for each data item so that an algorithm that is capable of handling users' fidelity requirements (as well as $tcr$s) can adapt to users' fidelity needs.

Traditionally the problem of maintaining cache consistency has been addressed either by server- or client-driven approaches. In client-driven approach, cache manager contacts the source periodically to check validity of the cached data. We call this period *Time-To-Refresh* or *TTR*. Choosing very small TTR values help in keeping cache consistent although at the cost of bandwidth. On the other hand, very large TTR values may reduce network utilization but only at the cost of reduced fidelity. *Polling-each-time* and *Adaptive TTR* are examples of client-driven techniques. Clearly these techniques are based on the assumption that an optimum TTR value can be predicted using some statistical information. This may not be true for highly dynamic data which is changing unpredictably and independently. The other class of algorithms are server-driven wherein server takes the responsibility of either invalidating or updating the proxy cache. Sending invalidation messages or pushing recent changes are examples of such techniques.

Also because of dynamics of the data, none of the above techniques can deliver high fidelity with optimum resource utilization. In the following sections we explain how one can use user specified constraints to offer high fidelity with efficient use of available resources.

# 3 The Pull Approach

The *Pull* approach is the most traditional approach for maintaining the temporal coherency of caches caching dynamic data. In this model, each data item is assigned a certain *TTR* when the data object is brought in the cache. Since the arrival of the data item and until time equal to $TTR$ elapses, all the requests for the data object are satisfied from the cache without looking up the values in the data sources. Thus in this approach, the proxy is responsible for obtaining the data from the server. The proxy issues a `GET` request to the server and the server just delivers the required data.

## 3.1 Periodic Pull

Most of the current applications do *WebCasting* [16] i.e., periodic polling as shown in figure 3.1. The user registers with the proxy which does "webcasting" with a constraint, the proxy periodically polls the server for this data periodically and whenever a change of user interest has occurred, it

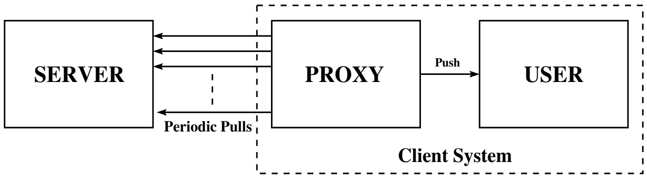


**Fig. 2.** Periodic Polling *aka* WebCasting

pushes the change to the user. This approach is equivalent to setting the *TTR* value of a cached item *statically*. Thus, the proxies obtain data from data sources with such a high frequency that the user gets the feel that the data is being pushed by the server only. However, this can lead to a very high network overheads in case the polling period is too low or may cause the user to miss some changes of interest if the polling period is too high. Clearly, this technique is useful only if rate of change of data is constant or relatively low (such as news). If the rate of change itself is varying, then this technique of assigning frequency apriori is not suitable (as in the case of stock quotes). But still, currently this is the most popular data delivery technique as it can be purely web-based (because of HTTP) and does not need any special resources (like push capability in servers or modification of HTTP).

### 3.2 Aperiodic Pull

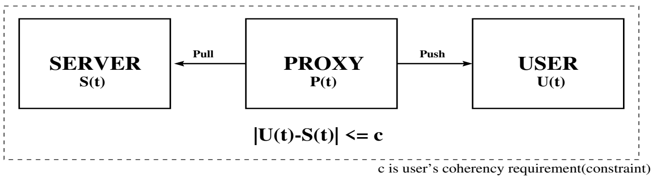


**Fig. 3.** Adaptive Polling

Since dynamic data changes independently and unpredictably, we cannot use standard prediction and forecasting algorithms for predicting the next $TTR$ value to be assigned to the data object. A new method for assigning $TTR$ values is given in [15]. Given a user's coherency requirement, this technique allows a proxy to adaptively vary the TTR value based on the rate of change of the data item. The TTR decreases dynamically when a data item starts changing rapidly and increases when a hot data item becomes cold. To achieve this objective, the *Adaptive TTR* approach takes into account (a) static bounds so that TTR values are not set too high or too low, (b) the most rapid changes that have occurred so far and (c) most recent changes to the polled data.

In what follows, we use $D_0$, $D_1, \ldots$, $D_l$ to denote the values of a data item $D$ at the server in chronological order. Thus, $D_l$ is the most recent value data item $D$.

The adaptive ***TTR*** is computed as:

$$\begin{aligned} TTR_{adaptive} = Max(&TTR_{min},\ Min(TTR_{max}, \\ &a \times TTR_{hr} + (1-a) \times TTR_{dyn})) \end{aligned}$$

where

- $[TTR_{min}, TTR_{max}]$ denote the range within which TTR values are bound.
- $TTR_{hr}$ denotes the most conservative, i.e., smallest, TTR value used so far. If the next TTR is set to $TTR_{hr}$, temporal coherency will be maintained even if the maximum rate of change observed so far recurs. However, this TTR is pessimistic since it is based on worst case rate of change at the source. If this worst case rapid change occur for only a small duration of time, then this approach is likely to waste a lot of bandwidth especially if the user can handle some loss of fidelity.
- $TTR_{dyn}$ is a learning based TTR estimate founded on the assumption that the dynamics of the last few (two, in the case of the formula below) recent changes are likely to be reflective of changes in the near future.

  $$TTR_{dyn} = (w \times TTR_{estimate}) + ((1-w) \times TTR_{latest})$$

  where
  - $TTR_{estimate}$ is an estimate of the TTR value, based on the most recent change to the data.

    $$TTR_{estimate} = \frac{TTR_{latest}}{|D_{latest} - D_{penultimate}|} \times c$$

    If the recent rate of change persists, $TTR_{estimate}$ will ensure that changes which are greater than or equal to $c$ are not missed.

- weight $w$ ($0.5 \leq w < 1$, initially 0.5) is a measure of the relative change between the recent and the old changes, and is adjusted by the system so that we have the *recency* effect, i.e., more recent changes affect the new $TTR$ more than the older changes.

– $0 \leq a \leq 1$ is a parameter of the algorithm and can be adjusted dynamically depending on the fidelity desired, with a higher fidelity demanding a higher value of $a$.

The adaptive TTR approach has been experimentally shown to have the best $tc$ properties among several TTR assignment approaches [15].

# 4 The Push Approach

In this method, the server is responsible for delivering the relevant data to the user. The server does not behave in *request-response* mode, where the server delivers some data only when there is an explicit request for it. But instead, server pushes the data into channel without any explicit request. As before, the server can push the data either periodically or aperiodically.

## 4.1 Periodic Push

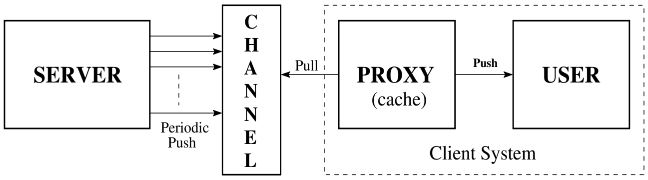


**Fig. 4.** Periodic Push (Channel acts like a data medium)

In this method (figure 3.1), the server is not aware of the exact coherency needs of the a particular client, but only of the general demand for data items. So, based on the general demand distribution of data items the server creates a schedule for dissemination data items. A data item with higher demand will be disseminated with higher frequency and vice-versa. All data items get divided into frequency bands, where data items belonging to one frequency band have similar demands. Once the push schedule is created using these frequency bands, it is not changed. The server then repeats this schedule periodically. The Broadcast Disks [1] approach 4.1 is one such approach where the frequency bands are termed as broadcast disks. This approach also provides for client feedback. An interesting property of this approach is that it treats the channel like a medium and tries to decide on the “format” in which the channel should hold the data. In a way the channel itself is acting like a proxy.

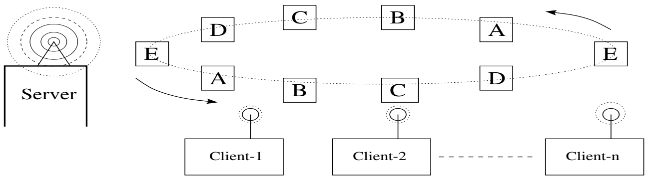


**Fig. 5.** Broadcast Disks

It is clear that since the server is not aware of the specific needs of the users, the push schedule may not be adequate for users who desire high fidelity and specific results. So, the approach may lead to low fidelity and/or wastage of bandwidth. But, the approach is useful for data which is generic and changes not very rapidly (e.g. news, digests, entertainment).

### 4.2 Aperiodic Push

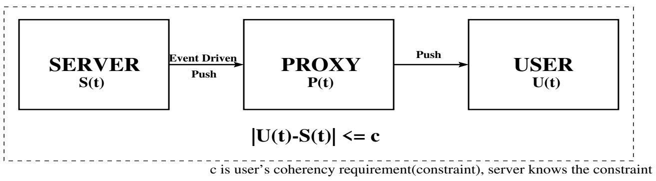


**Fig. 6.** Aperiodic Push

In the aperiodic push-based approach, the proxy registers with a server, identifying the data of interest and the associated $tcr$, i.e., the value $c$. Whenever the value of the data changes, the server uses the $tcr$ value $c$ to determine if the new value should be pushed to the proxy; only those changes that are of interest to the user (based on the $tcr$) are actually pushed (figure 4.2). Formally, if $D_k$ was the last value that was pushed to the proxy, then the current value $D_l$ is pushed if and only if $|D_l - D_k| \geq c$, $0 \leq k \leq l$. To achieve this objective, the server needs to maintain state information consisting of a list of proxies interested in each data item, the $tcr$ of each proxy and the last update sent to each proxy.

The key advantage of the this approach is that it can meet stringent coherency requirements—since the server is aware of every change, it can precisely determine which changes to push and when. A limitation of push-based servers is that the amount of state that needs to be maintained can be large, especially for popular data items. A server can optimize the state space overhead by combining requests from all proxies with identical $tcr$s into a single request; all proxies are notified if the change to the data item $D$ exceeds a specified $tcr$. Even with such optimizations, the state space overhead can be excessive, which in turn limits the scalability of the server. A further limitation of the approach is that it is not resilient to failures. The state information is lost if the server fails and requires the proxy to detect the failure and re-register its $tcr$ for the data item.

# 5 Push vs. Pull

Push and Pull approaches have complementary properties with respect to fidelity, network utilization, scalability and resiliency. We have summarized the properties in table 1.

## 5.1 Communication Overheads

In push-based approach, the number of messages transferred over the network is equal to the number of times the data changes so that the user specified temporal coherency is maintained. A pull-based approach requires two messages—a HTTP request, followed by a response—per poll. Moreover, in the pull approach, a proxy polls the server based on its estimate of how frequently the data is changing. If the data actually changes at a slower rate, then the proxy might poll more frequently than necessary. Hence a pull-based approach is liable to impose a larger load on the network. However, a push-based approach may push to clients who are no longer interested in a piece of information, thereby incurring unnecessary message overheads. The communication overhead also depends upon dynamics of the data. For rapidly changing data, in order to maintain high fidelity cache manager must poll the source very frequently. As the rate with which data is changing also varies with time, many of these requests may prove useless incurring unnecessary network load. Similarly if the data is changing very slowly then again many polls may prove useless.

**Table 1.** Properties of Push and Pull

| **Algorithm** | **Resiliency** | **Temporal Coherency** | **Overheads (Scalability)** | | |
|---|---|---|---|---|---|
| | | | *Communication* | *Computation* | *State Space* |
| Push | Low | High | Low | High | High |
| Pull | High | Low (for small constraints)<br>High (for large constraints) | High | Low | Low |

## 5.2 Computational Overheads

Computational overheads for a pull-based server result from the need to deal with individual pull requests. After getting a pull request from the proxy, the server has to just look up the latest data value and respond. On the other hand, when the server has to push changes to the proxy, for each change that occurs, the server has to check if the coherency requirement for any of the proxies has been violated. This computation is directly proportional to the rate of arrival of new data values and the number of unique temporal coherency requirements associated with that data value. Then the computational overhead per data item is of the order of rate of arrival of new values times the number of unique coherence requirements associated with that value. Although this is a time varying quantity in the sense that the rate of arrival of data values as well as number of connections change with time, it is easy to see that push is computationally more demanding than pull.

For each pull request, server has to open a new connection with the client and close it after the request is served. Opening/closing of connections clearly imposes a resource overload. The above observation may not hold if large number of clients are interested in similar data items.

In such cases, monitoring just one item can satisfy many client requirements i.e., the cost of monitoring that data item is amortized over a large number of clients. This cost is less than serving individual requests from each of the proxies. It makes sense to have push connections in such situations. In short, high computational load may arise either because of too much polling of the server or too much monitoring load.

### 5.3 Space Overheads

A pull-based server is stateless. In contrast, any push-based server must maintain the $c$ value for each client, the latest pushed value, along with the state associated with an open connection. Since this state is maintained throughout the duration of client connectivity, the number of clients which the server can handle may be limited when the state space overhead becomes large (resulting in scalability problems).

### 5.4 Resiliency

By virtue of being stateless, a pull-based server is resilient to failures. In contrast, a push server maintains crucial state information about the needs of its clients; this state is lost when the server fails. Consequently, the client's coherency requirements will not be met until the proxy detects the failure and re-registers the coherency requirements with the server.

In push-based techniques, we can classify failures as server side, client side or communication failure. Each of these has different implications on the behavior of the system.

– In case of server failures, state at the server is lost. Most of the push algorithms require state to be maintained at the server and hence their correctness may get compromised in such cases. Cache coherency is not guaranteed until the state is reconstructed at the server. It may not be possible for a server to initiate error recovery because it has no way to know which clients were being served when the crash occured. Client should somehow know about server failure so that it can start error recovery.
– Clients may also fail. A server has to allocate resources to each client. As resources are valuable, in case of unreachable clients these resources must be reclaimed. Push-based techniques rely on some kind

of feedback from the clients to handle clients failures. Obviously this may generate additional control messages adding to total communication overhead.

– Communication failures occur either due to socket failures at any one of the ends, network congestion or network partition. Push-based techniques must employ special mechanisms to deal with such errors.

## 5.5 Scalability

Pull servers are generally stateless and hence scalable. A Server has to respond to the incoming request, but need not maintain any state information or keep connection open with the client after the request is satisfied. Since open connections consume sockets and buffer space, it is necessary to close the unwanted connections. With an upper bound on the number of sockets and the state space available, this property is very desirable and often helps in making servers scalable.

Web servers deployed all over the world are pull-based and stateless. A user sends a request and waits for the response. The primary consideration has been to make the web servers scalable. It is true for normal applications, but for the data which is changing rapidly and that too with different rates, this may not be very true. There is certain overhead associated with opening and closing of connections. So the sockets once used may remain unavailable for some time period. When data at a source is changing very fast, the proxy will generate a large number of requests to keep its cache in sync with the source. Thus there will be a large overhead in opening and closing the connections. Also the computational load at the server becomes high because it has to respond to far more requests. The socket queues start filling up, increasing the response time and eventually a server may start dropping requests.

Push servers have complementary characteristics. The server has to keep sockets open and allocate enough buffers to handle each client. With large number of clients, state space and network resources can soon become bottlenecks and server may start dropping requests. In short, the scalability issue may arise because of the excessive server computation and network traffic or state space maintained at the server and resources allocated (such as sockets) and there is a clear tradeoff between these two constraints.

## 6 Need to Combine Push and Pull

From the above section it is clear that:

- A pull-based approach does not offer high fidelity when the data changes rapidly or when the coherency requirements are stringent (i.e., small values of $c$). Moreover, the pull-based approach imposes a large communication overhead (in terms of the number of messages exchanged) when the number of clients is large. But it may suffice for requests which have large values of $c$.
- A push-based algorithm can offer high fidelity for rapidly changing data and/or stringent coherency requirements. However, it incurs a significant computational and state-space overhead resulting from a large number of open push connections and their serving processes/threads. Moreover, the approach is less resilient to failures due to its stateful nature.

These properties indicate that a push-based approach is suitable when a client expects its coherency requirements to be satisfied with high fidelity, or when the communication overheads are a bottleneck. A pull-based approach is better suited to less frequently changing data or for less stringent coherency requirements, and when resilience to failures is important. The complementary properties of the two approaches indicate the need for having an approach which combines the advantages of both while not suffering from any of their disadvantages.

## 7 Combination of Push and Pull

As is clear from the previous discussion, neither push nor pull alone is sufficient for efficient dissemination of dynamic data. These two techniques have complimentary properties with respect to fidelity offered, network utilization, server scalability and resiliency. Few attempts have been made in the past to combine these two canonical techniques. *Adaptive leases* [8] and *Volume leases*[13] are two examples. The former is used for maintaining strong cache consistency in the World Wide Web while the later is used for caches holding a large number of data items. None of these is meant for highly dynamic data. Nor do they take user requirements into account. In this section we describe the Adaptive Leases approach. We also describe two algorithms that we have developed for better scalability and coherency for dynamic data.

### 7.1 Leases

*Leases are like contracts given to a lease holder over some property* [4]. Whenever some client requests server for a certain document, server returns that document along with a lease. In other words, a server takes the responsibility of informing the client about any changes during the lease period. Once a lease expires, a client must contact the server and renew the lease. Client can use the cached copy while it has a valid lease over the data item. During valid lease period, client remains in push mode and is switched back to pull mode after the lease expires. Thus the client is alternatively served in push and pull modes.

Clearly, pure leases are not very useful for dynamic data. It is very important to choose a good lease period. For a very high value, client remains in push mode for most of the time and scalability problem may arise. On the other hand, for small values the lease renewal cost may be prove very high. *Adaptive Leases* try to dynamically adjust the lease duration. The decision is based on many parameters like popularity of the data item, server state space available and network bandwidth available.

### 7.2 Dynamically Combining Push and Pull: PaP

In the PaP [7] approach, the proxy operates in pull mode using some TTR algorithm, while the server is in push mode and knows the constraint. Using this constraint and proxy access patterns the server tries to predict when a client is going to poll next. If it determines that within this predicted time the client is likely to miss a chhange of interest, it pushes that change to the client. For predicting the client connection times, the server may run the TTR algorithm in parallel with client or use some simpler approximation of it. Given network delays, a server waits for the client to pull within a small window around the predicted time. So, once the first few changes are intimitaed to the client (by pull or push), the rest of the successive changes will be known to the client easily.

In the ideal case, the fidelity offered will be 100%, but due to synchronization problems and other factors, it will be slightly less. But, it will always be much greater than pull. Because of the pull component the resiliency of the system will be high. And due to the push component, communication overheads will also be low. PaP also provides for fine tuning of its behavior. It has a few parameters which swing it towards more push or more pull, and thus its performance in terms of fidelity and resiliency can be controlled.

### 7.3 Dynamically Choosing Push or Pull: PoP

Another possibility is to divide incoming clients at the server into either push or pull clients and dynamically switch them to one or the other mode [7]. If resources are plentiful, every client is given a push connection irrespective of its fidelity requirements. This ensures that the best fidelity is offered. As more and more clients start requesting the service, resource contention may arise at the server leading to scalability problems. Few clients are then shifted to pull mode. Thus valuable resources are freed and system scales properly. Contrary to this, when resources again become available few high priority clients are switched back to push mode thus ensuring high fidelity.

The most important issue is how to assign priorities to different clients. Few of the possible parameters are the access frequency of each client, temporal coherency requirement, fidelity requirement and network bandwidth available. Clearly no single criterion suffices but collectively they have the potential to offer high average fidelity still keeping the system scalable.

## 8 Related Work

[2] is one of the earliest papers relating to the topic of maintaining coherency between a data source and cached copies of the data. This paper discusses techniques whereby data sources can propagate, i.e, push, updates to clients based on their coherency requirements. This paper also discusses techniques whereby cached objects can be associated with expiration times so that clients themselves can invalidate their cached copies.

More recently, various coherency schemes have been proposed and investigated for caches on the World Wide Web where the sources are typically pull-based and stateless. Thus, the source is unaware of users' coherency requirements and users pull the required data from the sources.

A *Weak consistency* mechanism, *Client polling*, is discussed in [3], where clients periodically poll the server to check if the cached objects have been modified. In the Alex protocol presented here, the client adopts an adaptive Time-To-Live(TTL) expiration time which is expressed as a percentage of the object's age. Simulation studies reported in [9] indicate that a weak-consistency approach like the Alex protocol ([3]) would be the best for web caching. The main metric used here is network traffic. While the Alex protocol uses only the time for which the source data remained unchanged, given our desire to keep temporal consistency within specified limits, we need to also worry about the magnitude of the change.

A *strong consistency* mechanism, *Server invalidation*, is discussed in [11], where the server sends invalidation messages to all clients when an object is modified. This paper compares the performance of three cache consistency approaches, and concludes that the invalidation approach performs the best.

A survey of various techniques used by web caches for maintaining coherence, including the popular "expiration mechanism", is found in [6]. It also discusses several extensions to this mechanism, but,as discussed in [15], these do not meet our needs.

Another approach is for the cache server to piggyback a list of cached objects [12] whenever it communicates with a server. The list of objects piggybacked are those for which the expiration time is unknown or the heuristically-determined TTL has expired.

## 9 Concluding Remarks

Since the frequency of changes of time-varying web data can itself vary over time (as hot objects become cold and vice versa), in this paper, we argued that it is *a priori* difficult to determine whether a push- or pull-based approach should be employed for a particular data item. Also, complicating the choice is the complementary properties relating to their resiliency as well as state-space and communication overheads. To address this limitation, we proposed two techniques that combine push- and pull-based approaches and adaptively determine which approach is best suited at a particular instant. We are currently evaluating the performance, functionality, and overhead profiles of these new algorithms so as to determine the range of their applicability for dissemianting dynamic Web data.

# References


1. S. Acharya, M. J. Franklin and S. B. Zdonik: Balancing Push and Pull for Data Broadcast, *Proceedings of the ACM SIGMOD Conference, May 1997*.
2. R. Alonso, D. Barbara, and H. Garcia-Molina: Data Caching Issuesin an Information Retrieval System. *ACM Trans. Database Systems, September 1990*.
3. A. Cate: Alex - A Global Filesystem. *Proceedings of the 1992 USENIX File System Workshop, Ann Arbor,MI May 1992*.
4. C. Gray and D. Cheriton: Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency, *Proceedings of the Twelfth ACM Symposium on Operating System Principles, pages 202-210,1989*.
5. P. Cao and S. Irani: Cost-Aware WWW Proxy Caching Algorithms., *Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997*.
6. A. Dingle and T. Partl: Web Cache Coherence. *Proc. Fifth Intnl. WWW Conference, May 1996*.
7. P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham and P. Shenoy: Adaptive Push-Pull of Dynamic Web Data: Better Resiliency, Scalability and Coherency. *Technical Report TR00-36, Department of Computer Science, University of Massachusetts at Amherst, July 2000*.
8. V. Duvvuri, P. Shenoy and R. Tewari: Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. *InfoCom, March 2000*.
9. J. Gwertzman and M. Seltzer: The Case for Geographical Push Caching., *Proceedings of the 5th Annual Workshop on Hot Operating Systems, pages 51-55, May 1995*.
10. A. Iyengar and J. Challenger: Improving Web Server Performance by Caching Dynamic Data., *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USEITS), December 1997*.
11. C. Liu and P. Cao: Maintaining Strong Cache Consistency in the World-Wide-Web., *Proceedings of the Seventeenth International Conference on Distributed Computing Systems, pages 12-21, May 1997*.
12. 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, December 1997*.
13. B. Krishnamurthy and C. Wills: Piggyback Server Invalidation for Proxy Cache Coherency. *Proceedings of World Wide Web Conference, April 1998*.
14. A. G. Mathur, R. W. Hall, F. Jahanian, A. Prakash and C. Rasmussen: The Publish/Subscribe Paradigm for Scalable Group Collaboration Systems., *Technical Report CSE-TR-270-95, Dept. of Computer Science and Engg., University of Michigan, 1995*.
15. Raghav Srinivasan, Chao Liang and Krithi Ramamritham: Maintaining Temporal Coherency of Virtual Data Warehouses, *The 19th IEEE Real-Time Systems Symposium (RTSS98), Madrid, Spain, December 2-4 1998*.
16. M. J. Franklin and S. B. Zdonik: ”Data In Your Face”: Push Technology in Perspective. *SIGMOD Conference, Seattle, Washington, May-June 1998*.