Adaptive Push-Pull: Disseminating Dynamic Web Data

Manish Bhide [*] Pavan Deolasee[†] Amol Katkar[‡] Ankur Panchbudhe [§] Krithi Ramamritham [¶] Prashant Shenoy[||]

March 8, 2002

Abstract

An important issue in the dissemination of time-varying web data such as sports scores and stock prices is the maintenance of *temporal coherency*. In the case of servers adhering to the HTTP protocol, clients need to frequently *pull* the data based on the dynamics of the data and a user's coherency requirements. In contrast, servers that possess *push* capability maintain state information pertaining to clients and push only those changes that are of interest to a user. These two canonical techniques have complementary properties with respect to the level of temporal coherency maintained, communication overheads, state space overheads, and loss of coherency due to (server) failures. In this paper, we show how to combine push- and pull-based techniques to achieve the best features of both approaches. Our combined technique tailors the dissemination of data from servers to clients based on (i) the capabilities and load at servers and proxies, and (ii) clients' coherency requirements. Our experimental results demonstrate that such adaptive data dissemination is essential to meet diverse temporal coherency requirements, to be resilient to failures, and for the efficient and scalable utilization of server and network resources.

Keywords: Dynamic Data, Temporal Coherency, Scalability, Resiliency, World Wide Web, Data Dissemination, Push, Pull

1 Introduction

Recent studies have shown that an increasing fraction of the data on the world wide web is time-varying (i.e., changes frequently). Examples of such data include sports information, news, and financial information such as stock prices. The coherency requirements associated with a 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 to have stronger coherency requirements for data items such as stock prices. 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.

In the rest of this section, we describe the problem of temporal coherency maintenance in detail, motivate the need to go beyond the canonical *Push*- and *Pull*-based data dissemination, and outline the key contributions of this paper, namely, the development and evaluation of adaptive protocols for disseminating dynamic i.e., time-varying data.

[*] Currently at IBM India Research Lab. Work done while at IIT Bombay.

[†] Currently at Veritas Software India Limited. Work done while at IIT Bombay and supported in part by a fellowship from IBM.

[‡] Currently at Veritas Software India Limited. Work done while at IIT Bombay and supported in part by a fellowship from IMC.

[§] Currently at Veritas Software India Limited. Work done while at IIT Bombay and supported in part by a fellowship from D. E. Shaw.

[¶] Contact author (krithi@iitb.ac.in). Currently at IIT Bombay. Work supported in part by Tata Consultancy Services' Center for Intelligent Internet Research at IIT Bombay and by NSF Grant CCR-0098060.

[||] Currently at the Univ. of Mass. Amherst. Work supported in part by NSF Grants CCR-9984030 and CCR-0098060, EMC, IBM, Intel, and Sprint.

1.1 The Problem of Maintaining Temporal Coherency on the Web

Suppose users obtain their time-varying data from a proxy cache. The caches that are deployed to improve user response times must track such dynamically changing data so as to provide users with temporally coherent information. To maintain coherency of the cached data, each cached item must be periodically refreshed with the copy at the server. We assume that a user specifies a *temporal coherency requirement* (tcr) for each cached item of interest. The value of tcr denotes the maximum permissible deviation of the cached value from the value at the server and thus constitutes the user-specified tolerance. Observe that tcr 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). In this paper, we only consider temporal coherency requirements specified in terms of the value of the object (maintaining temporal coherency specified in units of time is a simpler problem that requires less sophisticated techniques). As shown in Figure 1, let $S(t)$, $P(t)$ and $U(t)$ denote the value of the data item at the server, proxy cache and the user, respectively. Maintaining temporal coherency implies maintaining the following inequality for all times t:

$$\|U(t) - S(t)\| \leq c \tag{1}$$

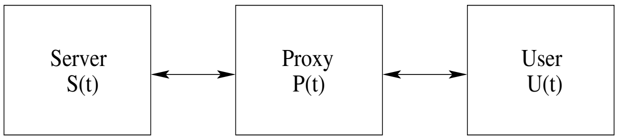

Figure 1: The Problem of Temporal Coherency

1.2 Contributions: Adaptive Algorithms for Disseminating Dynamic Data to Maintain Temporal Coherency

In the case of Web servers adhering to the HTTP protocol, proxies need to periodically *pull* the data based on the dynamics of the data and users' coherency requirements. In contrast, servers that possess *push* capability can maintain state information pertaining to clients and push only those changes that are of interest to a user/proxy. The first contribution of this paper is an extensive evaluation of the canonical push- and pull-based techniques using traces of real-world dynamic data. Our results, reported in Section 2, indicate that these two canonical techniques have complementary properties with respect to the level of temporal coherency maintained, scalability (measured in terms of communication overheads, state space overheads, and computation overheads), and resiliency to (server) failures. In particular, our studies show that a push-based approach is suitable when a client has stringent coherency requirements or when communication overheads are the bottleneck; A pull-based approach is better suited under less stringent coherency requirements, when server computational loads are the bottleneck, or when resilience to failures is important.

Since the temporal coherency desired by different users is likely to be different and the rate of change of dynamic data as well as the computation, communication, and state-space overheads incurred vary over time, it is difficult to statically choose between the push- and pull-based approaches – a static solution to the problem of disseminating dynamic, i.e., time-varying, data will not be responsive to client needs or load/bandwidth changes. We need an *intelligent* and *adaptive* approach that can be tuned according to the client requirements and conditions prevailing in the network or at the server/proxy. Moreover, the approach should not sacrifice the scalability of the server (under load) or reduce the resiliency of the system to failures. One solution to this problem is to combine push- and pull-based dissemination so as to realize the best features of both approaches while avoiding their disadvantages.

The development and evaluation of techniques that combine push and pull in an *intelligent* and *adaptive* manner while offering good resiliency and scalability forms the second contribution of this paper. We begin by offering two basic techniques for combining push- and pull-based dissemination:

1. PaP, our first algorithm, presented in Section 3, simultaneously employs both push and pull to disseminate data, but has tunable parameters to determine the degree to which push and pull are used. Conceptually, the proxy is primarily responsible for pulling changes to the data; the server is allowed to push additional updates that are undetected by the proxy. By appropriate tuning, our algorithm can be made to behave as a push algorithm, a pull algorithm or a combination. Since both push and pull are simultaneously employed, albeit to different degrees, we refer to this algorithm as *Push-and-Pull (PaP)*.

2. PoP, our second algorithm, presented in Section 4, allows a server to adaptively choose between push- and pull-based dissemination for each connection. Moreover, the algorithm can switch each connection from push to pull and vice versa depending on the rate of change of data, the temporal coherency requirements and resource availability. Since the algorithm dynamically makes a choice of push or pull, we refer to it as *Push-or-Pull (PoP)*.

Our third contribution is the development of enhancements – which offer more tuning knobs – to these basic combinations. In particular, Section 5 discusses two such enhanced protocols: (1) *PaPoP* which chooses one of PaP or Pull for each client thereby deploying the PaP combination for selected clients, and resorting to Pull for the rest. As in PoP, a PaP client in PaPoP may be dynamically converted to a Pull client depending on server resource contention, rate of change of data, and clients' temporal coherency requirements. (2) PaPoPL is a Lease-based [10, 12] variant of PaPoP. Here a PaP client can be converted to a Pull client only at the end of a lease period.

The overall contribution of this paper lies in the development of the the combined algorithms along with a systematic evaluation of these algorithms and comparing them to the two canonocal approaches. We have implemented our algorithms into a prototype server and a proxy. We demonstrate their efficacy via simulations and an experimental evaluation.

We end this section by noting that the goal of our research project is the development of practical dissemination algorithms for dynamic data. Intelligent and adaptive choice of these algorithms are the eventual goals of our current research and this paper studies various alternatives and their tradeoffs as an essential first-step towards this goal. To this end, we present a spectrum of approaches, from pure Pull and pure Push, through PoP and PaP, to PaPoP and PaPoPL, representing a repertoire of adaptive algorithms, each with its own special characteristics. This paper by itself does not provide detailed techniques for intelligent (and automatic) choice among the various algorithms, nor does it deal with the problem of tuning the parameter values automatically. But, for the sake of completeness, we briefly discuss issues relating to the tuning of these algorithms in Section 6 and compare and contrast these algorithms with the state of the art in Section 7.

Also, it is worth noting that our focus is on the path between a server and a proxy, assuming that push is used by proxies to disseminate data to end-users. A proxy can exploit user-specified coherency requirements by fetching and pushing only those changes that are of interest and ignoring intermediate changes. Since proxies act as immediate clients to servers, henceforth, we use the terms proxy and client interchangeably (unless specified otherwise, the latter term is distinct from the ultimate end-users of data).

2 Push vs. Pull: Algorithms and their Performance

In this section, we present a comparison of push- and pull-based data dissemination and evaluate their tradeoffs. These techniques will form the basis for our combined push-pull algorithms.

2.1 Pull

To achieve temporal coherency using a pull-based approach, a proxy can compute a *Time To Refresh (TTR)* attribute with each cached data item. The *TTR* denotes the next time at which the proxy should poll the server so as to refresh the data item if it has changed in the interim. A proxy can compute the *TTR* values based on the rate of change of the data and the user's coherency requirements. Rapidly changing data items and/or stringent coherency requirements result in a smaller TTR, whereas infrequent changes or less stringent coherency requirement require less frequent polls to the server, and hence, a larger *TTR*.[1] Observe that a proxy need not pull every single change to the data item, only those changes that are of interest to the user need to be pulled from the server (and the TTR is computed accordingly).

Clearly, the success of the pull-based technique hinges on the accurate estimation of the TTR value. Next, we summarize a set of techniques for computing the *TTR* value that have their origins in [22]. Given a user's coherency requirement, these techniques allow 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) the most recent changes to the polled data.

In what follows, we use D_0, D_1, ..., D_l to denote the values of a data item D at the server in chronological order. Thus, D_l is the latest value of data item D. $TTR_{adaptive}$ is computed as:
$Max(TTR_{min},\ Min(TTR_{max},\ a \times TTR_{mr} + (1-a) \times TTR_{dyn}))$ where

- $[TTR_{min}, TTR_{max}]$ denote the range within which TTR values are bound.
- TTR_{mr} denotes the most conservative, i.e., smallest, TTR value used so far. If the next TTR is set to TTR_{mr}, temporal coherency will be maintained even if the $m$aximum $r$ate 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 occurs 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 temporal coherency.
- 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 preference given to recent and 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 temporal coherency desired, with a stringent coherency demanding a higher value of a.

The adaptive TTR approach has been experimentally shown to have the best temporal coherency properties among several TTR assignment approaches [22]. Consequently, we choose this technique as the basis for pull-based dissemination.

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

2.2 Push

In a 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. 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 $|D_l - D_k| \geq c$, $0 \leq k < l$. 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 that proxy.

The key advantage of the push-based 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.

Next we discuss the dimensions along which the two approaches, as well as those we discuss subsequently, can be compared.

2.3 Comparing Dissemination Approaches: Data Fidelity and Associated Costs

We quantify the degree to which users' temporal coherency needs are met in terms of the *fidelity* of the data seen by users. We define fidelity f as the probability that the inequality (1) holds at any given time t.

Since a push-based server communicates every change of interest to a connected client, a client's tcr is never violated as long as the server does not fail or is so overloaded that the pushes are delayed. Thus, a push-based server is well suited to achieve a fidelity value of 1. On the other hand, in the case of a pull-based server, the frequency of the pulls (translated in our case to the assignment of TTR values) determines the degree to which client needs are met. We quantify the achievable fidelity of an algorithm as the ratio of the total length of time that the inequality 1 holds, that is, when $|U(t) - S(t)| \leq c$, to the total length of the observations. Let $\delta_1, \delta_2, \ldots, \delta_n$ denote these durations when user's tcr is violated. Let $observation\_interval$ denote the total time for which data was observed by a user. Then fidelity is

$$1 - \frac{\sum_{i=1}^{n} \delta_i}{observation\_interval}$$

and is expressed as a percentage. This then indicates the percentage of time when a user's desire to be within c units of the source is met.

In Section 2.4, we experimentally evaluate the fidelity of pull and push algorithms.

When we study the fidelity offered by an algorithm, we must also consider the scalability and resiliency properties of the associated algorithm. We study scalability of the server – to increasing numbers of clients or data items served – with respect to communication, computation, and space overheads of an algorithm. In this paper, we focus on the resiliency properties of an algorithm with respect to server failures since effects of communication failures and client failures affect push or pull based algorithms in the same way – a user does not get to observe updates of interest.

In the context of *scalability* of an algorithm, let us consider communication overheads first. We quantify *communication overheads* in terms of the number of messages exchanged between server and proxy. In a push-based approach, the number of messages transferred over the network is equal to the number of pushes, i.e., the number of times the user is informed of data changes so that the user specified temporal coherency is maintained. (In a network that supports multicasting, a single push message may be able to serve multiple clients.) A pull-based approach requires *two messages*—an HTTP IMS request, followed by a response—per poll. Clearly, given a certain number of updates received by a client, pull-based approaches incur larger message overheads. However, 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.

Turning to *computational overheads*, they can occur at the client, server, or both. 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 done to data item d to clients of d, for each change that occurs, the server has to check if the tcr for any of the clients has been violated. The latter might involve evaluating specific conditions. So,

$$comp\_overheads^d_{push}\ \alpha rate\_of\_updates(d)\ \times\ condition\_eval\_overhead\ \times\ \#unique\_tcr(d) \qquad (2)$$

This basically says that computational overheads are proportional to the rate at which d is updated, the overheads for evaluating the condition which determines if an update to d must be propagated to a client, and the number of tcrs associated with d. Although the above is a time varying quantity since the rate of updates as well as number of clients change with time, it is easy to see that push is computationally more demanding than pull, assuming that queueing related overheads for pull are negligible. Further, note that computation delays at the server can imply loss of fidelity at the clients since clients experience delayed data dissemination.

The final contributor to the (lack of) scalability is *space overhead*. A pull-based server is stateless. In contrast, a push-based server must maintain the tcr 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). To achieve a reduction in the space needed, rather than maintain the data and tcr needs of individual clients separately, the server combines all requests for a particular data item d and needing a particular tcr; as soon as the change to d is greater than equal to c, the tcr is specified for d, all the clients associated with d are notified. Let the above optimization process convert n connections into u unique (d, c) pairs. The state space needed is:

$$u \times (bytes\ needed\ for\ a\ (D, c)\ pair)\ + n \times (bytes\ needed\ for\ a\ connection\ state) \qquad (3)$$

Also, since $u \leq n$, this space is less than the space required if above optimization was not applied (in which case u in the first term of 3 will be replaced by n).

Finally, let us consider *resiliency*. If a server fails and a user expects the server to push changes of interest, a user might incorrectly believe that he has temporally coherent data and possibly take some action based on this. On the other hand, if a client explicitly pulls data from a server, it will soon enough, say at he end of a timeout period, know about the failure of the server and take appropriate action: In the case where there is just one server, if the user knows that that server has crashed, he/she can refrain from taking any decisions based on what he/she has. In case multiple servers are available, user can connect to a different server. Clearly, the latter could be done transparent to the actual user.

Another aspect of resiliency arises from loss of state at a server when it fails. By virtue of being stateless, a pull-based server is resilient to failures. In contrast, a push-based 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 tcr requirements with the server.

The above discussion clearly shows that pull is preferable when we consider the computational overheads, state space overheads and resiliency problems associated with push. On the other hand, the fidelity and communication overheads of pull depend on the nature of the data updates and the choice of TTRs. So we experimentally evaluate these two properties of Push and Pull next.

2.4 Experimental Evaluation of Push and Pull

In what follows, we compare the push and pull approaches with respect to maintenance of temporal coherency and communication overheads.

2.4.1 Experimental Model

These algorithms were evaluated using a prototype server/proxy that employed trace replay. For Pull, we used a vanilla HTTP web server with our prototype proxy. For Push, we used a prototype server that uses unicast and connection-oriented sockets to push data to proxies. All experiments were done on a local intranet. We also ran carefully instrumented experiments on the internet and the trends observed were consistent with our results. (A sample is discussed in the context of Figure 10.)

Note that it is possible to use multicast for push; however, we assumed that unicast communication is used to push data to each client (thus, results for push are conservative upper-bounds; the message overheads will be lower if multicast is used).

2.4.2 Traces Used

Quantitative performance characteristics are evaluated using real world stock price streams as exemplars of dynamic data. The presented results are based on stock price traces (i.e., history of stock prices) of a few companies obtained from `http://finance.yahoo.com`. The traces were collected at a rate of 2 or 3 stock quotes per second. Since stock prices only change once every few seconds, the traces can be considered to be "real-time" traces. For empirical and repetitive evaluations, we "cut out" the history for the time intervals listed in Table 1 and experimented with the different mechanisms by determining the stock prices they would have observed had the source been live. A trace that is 2 hours long, has approximately 15000 data values. All curves portray the averages of the plotted metric over all these traces. Few of the experiments were done with quotes obtained in real-time, but the difference was found to be negligible when compared to the results with the traces.

The Pull approach was evaluated using the Adaptive TTR algorithm with an a value of 0.9, TTR_{min} of 1 second and three TTR_{max} values of 10, 30 and 60 seconds. In the following sections, we evaluate how well an

Table 1: Traces used for the Experiment

Company	Date	Time	Maximum Value	Minimum Value	Average Value
Dell	Jun 1, 2000	21:56-22:53 IST	43.75	42.875	43.434932
UTSI	Jun 1, 2000	22:41-23:15 IST	22.25	21.00	21.731243
CBUK	Jun 2, 2000	18:31-21:57 IST	8.625	8.25	8.505309
Intel	Jun 2, 2000	22:14-01:42 IST	134.500	132.500	133.462484
Cisco	Jun 6, 2000	18:48-22:20 IST	65.000	63.0625	63.971584
Oracle	Jun 7, 2000	00:01-01:59 IST	79.3750	76.6250	78.576186
Veritas	Jun 8, 2000	21:20-23:48 IST	137.000	133.500	134.859644
Microsoft	Jun 8, 2000	21:02-23:48 IST	69.6250	68.0781	69.046370

algorithm is able to disseminate data, given c, the coherency requirement specified by a user. We examine a range of absolute c values, from fairly stringent requirements, represented by a c value of \$0.05 (likely to be important in currency trading and institutional investing scenarios), to looser requirements, represented by a c value which is an order of magnitude higher (likely to be sufficient to keep track of broad market trends).

2.4.3 Fidelity

Figure 2 shows the fidelity for a pull-based algorithm that employs adaptive *TTRs* . Recall that the Push algorithm offers a fidelity of 100%. In contrast, the Figure shows that the *pull* algorithm has a fidelity of 70-80% for stringent coherency requirements and its fidelity improves as the coherency requirements become less stringent. (The curve marked PaP is for the *PaP* algorithm that combines Push and Pull and is described in Section 3.2).

2.4.4 Communication Overheads

Figure 3 shows the variation in the number of messages with coherence requirement $\$0.05 \leq c \leq \$0.4$. As seen in Figure 3, the *Push* approach incurs a small communication overhead because only values of interest to a client are transferred over the network. The *Pull* approach, on the other hand, imposes a significantly higher overhead.

The above results indicate that A pull-based approach does not offer high fidelity when the coherency requirements are stringent. Moreover, the pull-based approach imposes a large communication overhead (in terms of the number of messages exchanged) than push.

Our discussions in Section 2.3 along with the above results are summarized in Table 2.

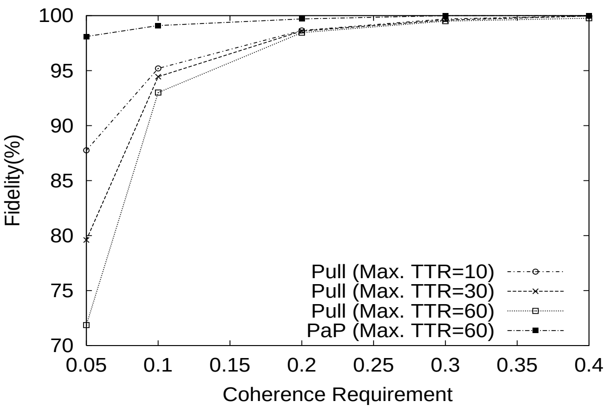

Figure 2: Effect of Varying Coherence Requirements on Fidelity

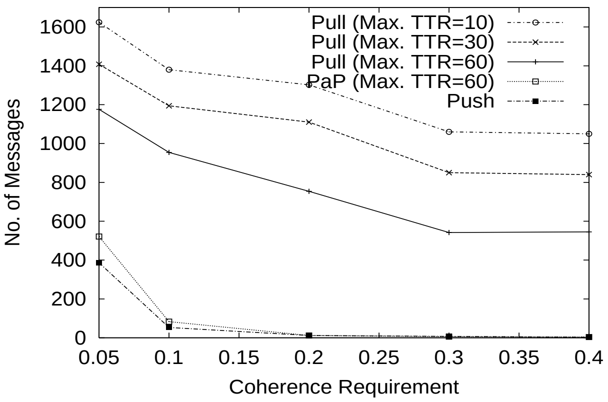

Figure 3: Effect of Varying Coherence Requirements on Communication Overheads

In what follows, we present two approaches that strive to achieve the benefits of the two complementary approaches by adaptively combining Push and Pull.

3 PaP: Dynamic Algorithm with Push and Pull Capabilities

In this section, we present *Push-and-Pull (PaP)* — a new algorithm that simultaneously employs both push and pull to achieve the advantages of both approaches. The algorithm has tunable parameters that determine the degree to which push and pull are employed and allow the algorithm to span the entire range from a push approach to a pull approach.

3.1 Motivation

The pull-based adaptive TTR algorithm described in Section 2.1 can react to variations in the rate of change of a data item. When a data item starts changing more rapidly, the algorithm uses smaller TTRs (resulting in more frequent polls). Similarly, if the changes are slow, TTR values tend to get larger. If the algorithm detects a violation in the coherency requirement (i.e., $|D_{latest} - D_{penultimate}| > c$), then it responds by using a smaller TTR for the next pull. A further violation will reduce the TTR even further. Thus, successive violations indicate that the data item is changing rapidly and the proxy gradually decreases the TTR until the TTR becomes sufficiently small to

Table 2: Characteristics of Push and Pull

Algorithm	Fidelity	Overheads (Scalability)			Resiliency
		Communication	*Computation*	*State Space*	
Push	High	Low	High	High	Low
Pull	Low (for small tcrs) High (for large tcrs)	High	Low	Low	High

keep up with the rapid changes. Experiments reported in [22] show that the algorithm gradually "learns" about such "clubbed" (i.e, successive) violations and reacts appropriately. So, what we need is a way to prevent even the small number of temporal coherency violations that occur due to the delay in this gradual learning process. Furthermore, if a rapid change occurs at the source and then the data goes back to its original value before the next pull, this "spike" will go undetected by a pull-based algorithm. The PaP approach described next helps the TTR algorithm to "catch" all the "clubbed" violations properly; moreover "spikes" also get detected. This is achieved by endowing push capabilities to servers and having the server push changes that a proxy is unable to detect. This increases the fidelity for clients at the cost of endowing push capability to servers. Note that, since proxies continue to have the the ability to pull, the approach is more resilient to failures than a push approach (which loses all state information on a failure).

3.2 The PaP Algorithm

Suppose a client registers with a server and intimates its coherency requirement tcr. Assume that the client pulls data from the server using an algorithm, say A, to decide its TTR values (e.g., *Adaptive TTR*). After initial synchronization, server also runs algorithm A. Under this scenario, the server is aware of when the client will be pulling next. With this, whenever the server sees that the client must be notified of a new data value, the server pushes the data value to the proxy if and only if it determines that the client will take time to poll next. The state maintained by this algorithm is a soft state in the sense that even if push connection is lost or the clients' state is lost due to server failure, the client will continue to be served at-least as well as under A. Thus, compared to a Push-based server, this strategy provides for graceful degradation.

In practice, we are likely to face problems of synchronization between server and client because of variable network delays. Also, the server will have the additional computational load imposed by the need to run the TTR algorithm for all the connections it has with its clients. The amount of additional state required to be maintained by the server cannot be ignored either. One could argue that we might as well resort to Push which will have the added advantage of reducing the number of messages on the network. However, we will have to be concerned with the effects of loss of state information or of connection loss on the maintenance of temporal coherency.

Fortunately, for the advantages of this technique to accrue, the server need not run the full-fledged TTR algorithm. A good approximation to computing the client's next TTR will suffice. For example, the server can compute the difference between the times of the last two pulls ($diff$) and assume that the next pull will occur after a similar $delay$, at $t_{predict}$. Suppose $T(i)$ is the time of the most recent value. The server computes $t_{predict}$, the next predicted pulling time as follows:

- let $diff = T(i) - T(i-1)$
- server predicts the next client polling time as $t_{predict} = T(i) + diff$.

If a new data value becomes available at the server before $t_{predict}$ and it needs to be sent to the client to meet the client's tcr, the server pushes the new data value to the client.

In practice, the server should allow the client to pull data if the changes of interest to the client occur close to the client's expected pulling time. So, the server waits, for a duration of ϵ, a small quantity close to TTR_{min}, for the client to pull. If a client does not pull when server expects it to, the server extends the push duration by adding $(diff - \epsilon)$ to $t_{predict}$. It is obvious that if $\epsilon = 0$, *PaP* reduces to push approach; if ϵ is large then the approach works similar to a pull approach. Thus, the value of ϵ can be varied so that the number of pulls and pushes is balanced properly. ϵ is hence one of the factors which decides the temporal coherency properties of the PaP algorithm as well as the number of messages sent over the network.

3.3 Details of PaP

The arguments at the beginning of this section suggest that it is a good idea to let the proxy pull when it is polling frequently anyway and violations are occurring rapidly. Suppose, starting at t_i a series of rapid changes occurs to data D. This can lead to a sequence of "clubbed" violations of tcr unless steps are taken. The adaptive TTR algorithm triggers a decrease in the TTR value at the proxy. Let this TTR value be TTR_k. The proxy polls next at $t_{i+1} = t_i + TTR_k$. According to the PaP algorithm, the server pushes any data changes above tcr during (t_i, t_{i+1}). Since a series of rapid changes occurs, the probability that some violation(s) may occur in $(t_i, t_{i+1}]$ is very high and thus these changes will also be pushed by the server further forcing a decrease in the TTR at the proxy and causing frequent polls from the proxy. Now, the TTR value at the proxy will tend towards TTR_{min} and $diff$ will also approach zero, thus making the durations of possible pushes from the server close to zero. It is evident that if rapid changes occur, after a few pushes, the push interval will be zero, and client will pull almost all the rapid changes thereafter. Thus the server has helped the proxy pull sooner than it would otherwise. This leads to better fidelity of data at the proxy than with a pull approach.

If an isolated rapid change (i.e., spike) occurs, then the server will push it to the proxy leading to a decrease in the TTR used next by the proxy. It will poll sooner but will not find any more violations and that in turn will lead to an increase in the TTR.

Thus, the proxy will tend to pull nearly all but the first few in a series of rapid changes helped by the initial pushes from the server, while all "spikes" will be pushed by the server to the proxy. The result is that all violations will be caught by the PaP algorithm in the ideal case (e.g., with the server running the adaptive TTR algorithm in parallel with the proxy). In case the server is estimating the proxy's next TTR, the achieved temporal coherency can be made to be as close to the ideal, as exemplified by Pure Push, by proper choice of ϵ.

Overall, since the proxy uses the pushed (as well as pulled) information to determine TTR values, the adaptation of the TTRs would be much better than with a pull-based algorithm alone.

Although the amount of state maintained is nearly equal to push, the state is a soft state. This means that even if the state is lost due to some reason or the push connection with a proxy is lost, the performance will be at-least as good as that of TTR algorithm running at the proxy as clients will keep pulling.

3.4 Performance of PaP

Figure 2 shows that for *PaP* algorithm, the fidelity offered is more than 98% for stringent tcr and 100% for less stringent tcr. From Figure 3, we see that compared to Pull, the *PaP* algorithm has very little network overhead because of the push component. Its network overheads are, however, slightly higher than that of *Push*.

The value of TTR_{max} and ϵ needs to be chosen to balance the number of pushes and pulls. The results of varying TTR_{max} and ϵ for a given value of c are shown in Figures 4 and 5.

Figure 4 shows that when ϵ is zero, PaP reduces to push and hence fidelity is 100%. But as we start increasing the value of ϵ the fidelity starts suffering. For $\epsilon = TTR_{min} = 1$, fidelity is approximately 99%. For values of ϵ closer to TTR_{max}, fidelity is low as the pulls overtake pushes and the algorithm behaves like a pure Pull algorithm. Figure 4 also shows that as TTR_{max} decreases, pulls increase. As pulls become more dominant, the server has less chance to push the data values, and a bigger ϵ gives the server fewer opportunities to push. This explains the effect in Figure 4 for $TTR_{max} = 5$ or $TTR_{max} = 10$. As pulls increase and the server has less and less chance to push, fidelity suffers and decreases more rapidly than in the case of $TTR_{max} = 60$. It can also be observed that, as ϵ takes values greater than TTR_{max}, fidelity offered becomes constant. This is because even if server sets ϵ greater than TTR_{max} client will keep polling at the maximum rate of TTR_{max}. In effect, setting ϵ greater than TTR_{max} is equivalent to setting it to TTR_{max}. This explains the crossover of curves in Figure 4.

Figure 5 also portrays the variation in the percentage of pushes and pulls. As expected, as ϵ is increased the percentage of pulls become higher and higher. For $\epsilon = 0$, there are no pulls and for $\epsilon = TTR_{max}(= 60)$ there are no pushes. If we compare the graphs in Figure 5, we can see that more fidelity requires more pushes and for

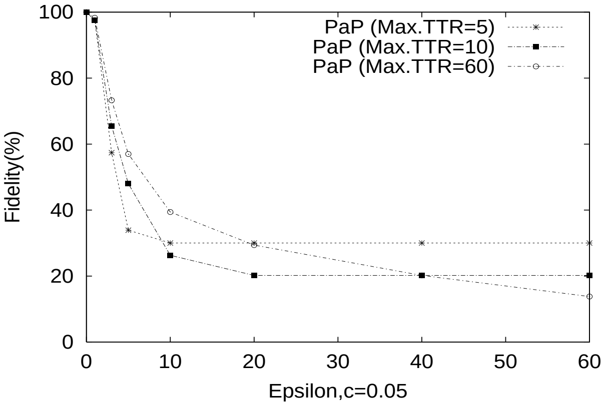

Figure 4: Effect of epsilon on PaP's Fidelity – c=0.05

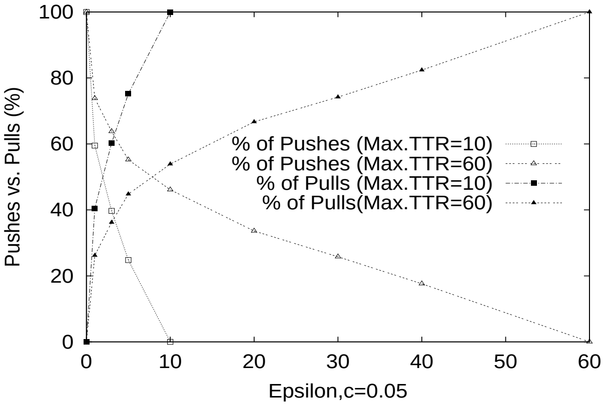

Figure 5: Effect of epsilon on PaP's Overheads – c=0.05

the case where percentage of pushes equals pulls, fidelity is close to 50%. The more we resort to pulls (i.e., with larger ϵ), the lower the obtained fidelity. The graphs for $TTR_{max} = 10$ in Figure 5 show a similar trend to those for $TTR_{max} = 60$, over the values $\epsilon = 0$ to $\epsilon = 10$ and 60 respectively. This is expected, as ϵ takes values between TTR_{min} and TTR_{max}, the behavior of the algorithm changes from being like pure push to pure pull (i.e., percentage of pushes drops). Setting $\epsilon = TTR_{max}$ makes the percentage of pushes equal zero.

4 PoP: Dynamically Choosing Between Push or Pull

PaP achieves its adaptiveness through the adjustment of parameters such as ϵ and TTR_{max}, and thereby obtains a range of behaviors with push and pull at the two extremes. We now describe a somewhat simpler approach wherein, based on the availability of resources and the data and temporal coherency needs of users, a server chooses push or pull for a particular client. Consequently, we refer to our approach as *Push-or-Pull (PoP)*.

4.1 Motivation

The resources that a server has to devote for a client connection are sockets, memory, CPU time and the bandwidth it has to make available for the connection. The space and time taken per push or pull connection can be estimated and from this, it is possible to determine if a certain combination of pulls and pushes can be sustained by a server given the memory, bandwidth and CPU resources allocated to it. It would be preferable to allow the server to set its allocation (to different types of services) dynamically so as to maximize the resource utilization as also the number of clients served given their requirements regarding desired fidelity.

4.2 The PoP Algorithm

PoP is based on the premise that at any given time a server can categorize its clients either as push clients or pull clients and this categorization can change with system dynamics. This categorization is possible since the server knows the parameters like the number of connections it can handle at a time and can determine the resources it has to devote to each mode (Push/Pull) of data dissemination so as to satisfy its current clients. The basic ideas behind this approach are:

- allow failures at a server to be detected early so that, if possible, clients can switch to pulls, and thereby achieve graceful degradation to such failures. To achieve this, servers are designed to push data values to their push clients when one of two conditions is met: (1) The data value at the server differs from the previously forwarded value by tcr or more. (2) A certain period of time TTR_{limit} has passed since the last change was forwarded to the clients. The first condition ensures that the client is never out of sync with the values at the server by an amount exceeding the tcr of the client. The second condition assures the client after passage of every TTR_{limit} interval that (a) the server is still up and (b) the state of the client with the server is not lost. This makes the approach resilient. In case of the state of the client being lost or the connection being closed because of network errors, the client will come to know of the problem after TTR_{limit} time interval, after which the client can either request the server to reinstate the state or start pulling the data itself. This ensures that in the worst case, the time for which the client remains out of sync with the server never exceeds TTR_{limit}.
- In this approach, the server can be designed to provide push service as the default to all the clients provided it has sufficient resources.
- When a resource constrained situation arises (upon the registration of a new client or network bandwidth changes) some of the push-based clients are converted to become pull-based clients based on the criteria that we had determined earlier.

Figure 6 gives the state diagram for achieving this adaptation.

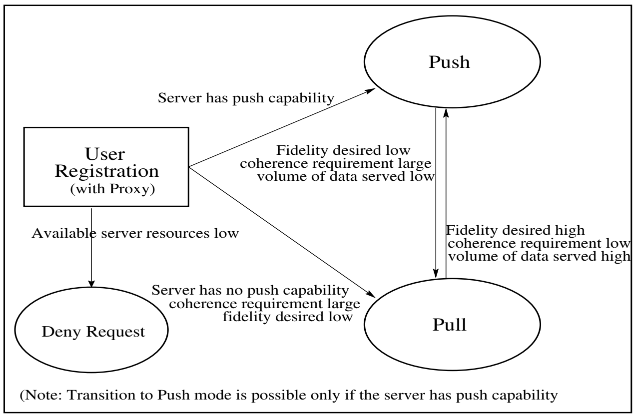

Figure 6: PoP: Choosing between Push and Pull

4.3 Details of PoP

Whenever a client contacts a server for a data item, the client also specifies its tcr and fidelity requirements.

- Irrespective of the fidelity requirement, if the server has sufficient resources (such as a new monitoring thread, memory, etc.), the client is given a push connection.
- Otherwise, if the client can tolerate lower fidelity, then server disseminates data to that client based on client pull requests.
- If the request desires 100% fidelity and the server does not have sufficient resources to satisfy it, then the server takes steps to convert some push clients to pull. If this conversion is not possible, then the new request is denied.

In the last case, the push clients chosen are those who can withstand the resulting degraded fidelity, i.e., those who had originally demanded less than 100% fidelity but had been offered higher fidelity because resources were available then for push connections. Which client(s) to choose is decided based on additional considerations including (a) bandwidth available (b) rate of change of data and (c) tcr . If bandwidth available with a client is low, then forcing the client to pull will only worsen its situation since pull requires more bandwidth than push. If the rate of change of data value is low or the tcr is high, then pull will suffice. Thus, from amongst the clients which had specified low fidelity requirement, we choose proxies which have (a) specified a high value of tcr, or (b) volume of data served is small. If a suitable (set of) client(s) is found, the server sends appropriate "connection dropped" intimation to the client so that it can start pulling.

4.4 Performance of PoP

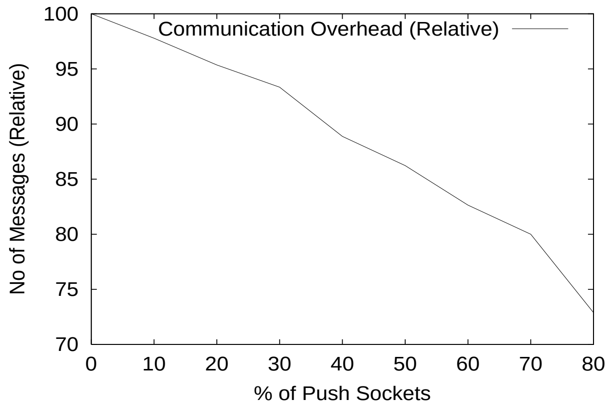

Figure 7: Effect of % of Push Connections on Message Overheads

Using the same traces given in table 1 we evaluated PoP. The experiments were performed by running the server on a lightly loaded Linux workstation and simulating clients from four different lightly loaded workstations. There were 56 users on each client machine, each accessing 3-4 data items. Keeping the server's communication resources constant, the ratio of push to pull connections was varied and the effect on average fidelity experienced by clients in pull mode as well as across all the clients was measured. Experimental results plotted in Figure 7 indicate that the communication overheads drop when the percentage of push sockets is increased. This is to be expected because push is optimum in terms of communication overheads. As we increase the percentage of push connections, while the push clients may be able to experience 100% coherence, as discussed next, the percentage of pull requests that are denied due to queue overflows grows considerably. These results indicate that a balance between pull and push connections must be struck if we want to achieve scalability while achieving high coherency.

Request denials are portrayed in detail in Figure 8. It shows that as the percentage of push connections is increased beyond 40%, request denials increase substantially. At the server, queues are maintained with each pull connection. When a new pull request is received at the server and there is no space in the queues, the request is

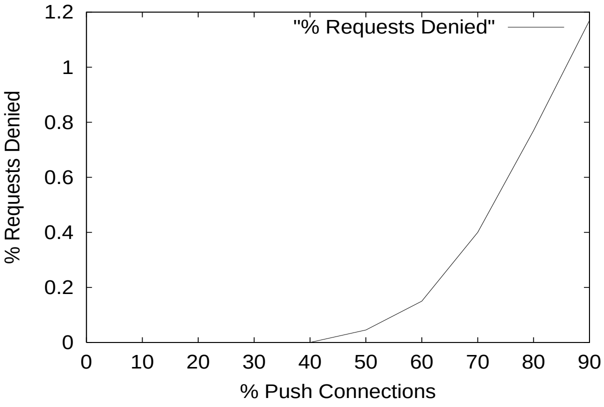

Figure 8: Effect of %Push Connections on Request Denials

denied. For these experiments, the queue size per connection was set to 4 and the total number of connections was set to 20. So, when 40% of the connections are allocated for pushes, 12 are available for pulls – with the potential to have 48 outstanding requests. The 49^{th} outstanding request will be denied. When a client receives a NACK (corresponding to a denied request) it contacts the server after a time interval equal to the current TTR. Request denials hence cause an increase in the effective response times for pull clients and hence overall system fidelity decreases.

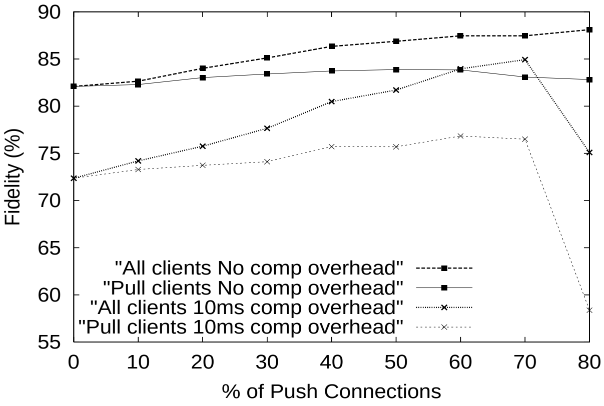

Figure 9: Effect of %Push Connections on PoP's Fidelity

Next, we discuss the effect of increasing the percentage of push connections on fidelity. As the number of push connections increases, proxies which were serving the largest number of data items or data items with stringent temporal coherency requirements are moved from pull to push mode[2]. The results are plotted in Figure 9 for two cases of computational overheads per pull request: (1) no computational overheads, other than those connected with the use of the socket, and (2) a 10 msec computational overhead per pull request, in addition to the socket handling overheads.

When the computational overheads for a pull are negligible, average fidelity across all clients improves gradually as we increase the percentage of push clients. When a small computational overhead of 10 msecs per pull is added, while fidelity improves up to a point, when the number of available connections for pull clients becomes

[2]The implemented system worked with a fixed number of clients/data items and so these results do not reflect the effect of admission control (i.e., request denial based on server load and client profile, which includes its requirements, volume of data served to it and its network status) that is part of PoP.

small, some of the pull requests experience denial of service thereby affecting the average fidelity across all clients. In fact, the overall fidelity drops nearly 10%.

Recall that all push clients experience 100% fidelity. So, the above drop in fidelity is all due to the pull clients. This is clear when we study the variation of the average fidelity of pull clients. With zero computational overheads for pulls (as has been the case upto now), as we increase the number of push clients, fidelity for pull clients improves from 82% to to 84% before dropping to 83%. The improvement as well as drop is more pronounced under 10 msec pull overheads. When a large percentage of the clients are in pull mode, the number of pull requests is very high. This increases the average response time for each client, which in effect, decreases the fidelity for pull clients. This scalability problem is due to computation load at the server when a large number of pull clients are present. As more and more clients switch to push mode, the number of pull requests drops, the response time of the server improves, and better fidelity results. But, the fidelity for pull clients peaks and then starts deteriorating. At this point the incoming requests cause overflows in the queues associated with pull connections and the corresponding requests are denied.

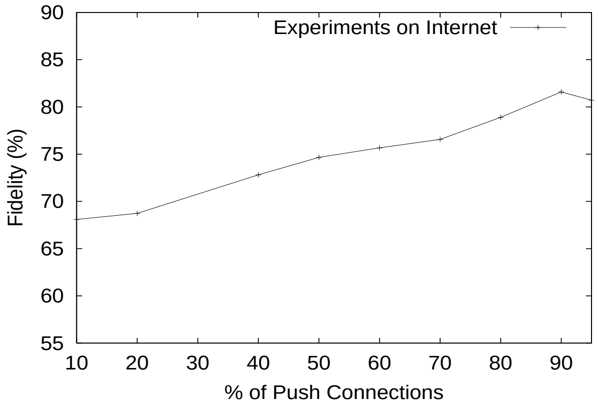

Figure 10: Effect of %Push Connections on PoP's Fidelity

We wanted to investigate whether the trends observed so far are also observed when the server is on the internet (in contrast with the controlled intranet environment used in the experiments so far). Figure 10 shows the variation in average fidelity with the % of push connections when the experiments were run on the internet. A comparison of Figures 10 with Figure 9 shows that the trends seen on the internet are exactly same as that seen in our intranet environment. The last portion of the curves in these figures clearly bring out the scalability issue arising because of resource constraints. These results identify the need for the system to allocate push and pull connections intelligently. An appropriate allocation of push and pull connections to the registered clients will provide the temporal coherency and fidelity desired by them. In addition, when clients request access to the server and the requisite server resources are unavailable to meet needs of the client, access must be denied. As Figure 6 indicates, this is precisely what PoP is designed to do.

5 Beyond PaP and PoP

We now present two extensions of PoP and PaP: In the first, PoP and PaP are combined: Specifically, we start with PoP and replace some of PoP's Push connections with PaP; In the second, we discuss a lease-based variant of the above.

5.1 PaPoP: PaP added to PoP

PaPoP is a combination of the PoP and PaP approaches:

Table 3: Average Fidelity and Bandwidth Utilization of PaPoP

#Total Clients	#Pull Clients	PoP			PaPoP			Drop in Fidelity	%age Increase in Msg Overhead
		Avg Fidelity	*#Pulls*	*#Pushes*	*Avg Fidelity*	*#Pulls*	*#Pushes*		
224	176	87.50	15037	10462	83.03	21227	5434	4.47	18.14
224	152	90.11	12000	15010	83.15	20700	8500	6.96	27.91
224	128	92.03	9917	18270	84.50	19800	10987	7.53	32.76
224	104	93.40	7774	19717	84.60	18560	12050	8.80	39.43

- Push clients of PoP are replaced with PaP clients: (a) the average resiliency offered would be better than PoP, and (b) the degradation in service is also also likely to be more graceful.
- Integration of PaP with PoP makes the approach more adaptive (by fine tuning using the parameters of PaP).
- The PoP algorithm at the server improves the average scalability offered by the system.

Since the resource requirements for Push as well as PaP are almost the same, the load at the server is the same as that of PoP. It should be noted that PaP generates pull requests (because server allows client to pull for ϵ time), which were absent in PoP. These pull requests are in addition to the requests generated by Pull clients and hence the load on the server may increase. But, as long as ϵ is small or the number of PaP clients is small as compared to Pull clients, this load is negligible and we get results similar to that of pure PoP.

As the proportion of pure-pull clients decreases, the extra load generated by replacing push with PaP also increases. For a very high ratio of PaP clients to pull clients, the number of extra pull requests generated by the PaP clients can be quite high. This leads to higher (pull) request denial rate, and fidelity can be compromised. Thus, the improvement in resiliency is achieved at the extra cost of a higher bandwidth utilization, higher server load and reduced fidelity.

To understand the fidelity and overhead characteristics of PaPoP, we conducted experiments by replacing the push component of a PoP server with PaP and compared them against PoP. The first column in table 3 corresponds to the total number of clients which is kept constant and the second column indicates the number of pure-pull clients. The last two columns summarize the results. The drop in fidelity refers to the absolute reduction in the average fidelity offered by PaPoP compared to the average fidelity offered by PoP: Drop in fidelity = Fidelity of PoP - Fidelity of PaPoP. (Recall that fidelity itself is computed as percentage.) Increase in message overhead is computed as a percentage increase in the number of messages for PaPoP beyond that needed for PoP. Similarly, the increase in message overhead indicates the percentage increase in the communication overhead (counting each pull as 2 messages and each push as 1 message) with PaPoP relative to PoP.

PoP has better fidelity and lower message overheads compared to PaPoP as the proportion of Pure-Pull clients decreases (i.e., as we go down the table). These results are quite intuitive. In PoP, for a large number of push sockets, most of the clients enjoy push service. So the average fidelity offered and communication overheads are optimum (recall that push offers optimum fidelity). When these push clients are shifted to PaP, the fidelity drops slightly. Also, because PaP also generates pull requests, communication overheads increase. These added pull requests load the server, as a result of which fidelity decreases. In summary, while PaPoP has the potential to offer higher resiliency, this comes at the cost of drop in fidelity and increase in the communication overheads.

Together, these arguments motivate the properties of PaPoP mentioned in Table 4: by adaptively choosing pull or PaP for its clients, PaPoP can be designed to achieve the desired temporal coherency, scalability and resiliency desired for a system.

5.2 PaPoPL: Adding Leases to PaPoP

In the leases approach [10, 12], the server agrees to push updates to a proxy so long as the lease is valid; the proxy must pull changes once the lease expires (or renew the lease). Thus, the technique employs push (during the lease) followed by pull and hence has higher resiliency than Push. The leases approach has high fidelity so long as the lease is valid and then has the fidelity of pull until the lease is renewed. The PaP approach has even greater resiliency than pure leases, since proxies continue to pull even if the server stops pushing.

Clearly, the leases approach can be combined with the PaPoP algorithm — the lease duration then indicates the duration for which the server agrees to provide a PaP connection.

In the rest of this Section, we present the details of the combination of PaPoP with leases along with a performance evaluation.

In PaPoPL, the server assigns a PaP connection to the proxy as long as the lease is valid; the proxy either renews the lease or resorts to using pull when the lease expires. When a proxy contacts the server it tries to acquire a lease from the server. Since the granting and maintenance of leases incurs space overheads at the server, it assigns leases to only a few of its clients. A server handles a limited number of connections and it gives a fixed percentage of these connections as PaP connections with leases. Since the message overhead in case of leases is less than that of Pull, out of its current connections, the server assigns leases to those connections that need more messages (and hence a larger network overhead) to maintain coherence. The proxy can keep track of the network overhead for each client and every time it contacts the server to establish a connection, it sends this value to the server. Hence this requires no extra state information at the server. The lease duration is decided by the server as a fixed *multiplication factor (MF)* of the last TTR for that data item (this can also be sent by the proxy to the server).

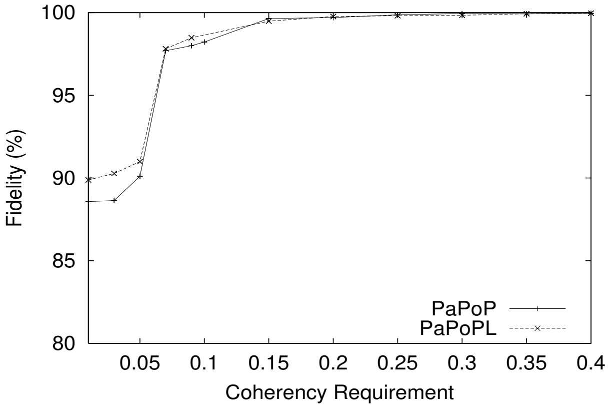

Figure 11: Comparison of Fidelity for PaPoP and PaPoPL

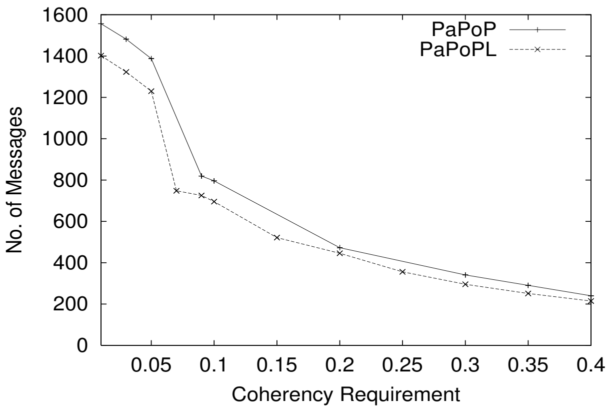

Figure 12: Comparison of Message Overheads for PaPoP and PaPoPL

Figures 11 and 12 clearly shows the advantage of PaPoPL over PaPoP. In these experiments the queue size for each connection was infinite. As a result there was no denial of request for pull connection. The computation overhead for pull request was set to zero and the ϵ value was set to 2. MF was set to 16. The plots show that with leases support, PaPoPL scores over PaPoP in terms of fidelity as well as communication overheads. In PaPoPL every client has to renew the lease at the end of the lease duration. This enables the algorithm to alter its behaviour according to the current trend in the data. The figures support this fact with PaPoPL offering better fidelity for strict coherency requirements. For looser coherency requirements, the performance of both the algorithms is nearly same as the chances of the data changing by such large values is very small. Nevertheless, for all coherency values, the message overheads under PaPoPL is about 10%lower than with PaPoP.

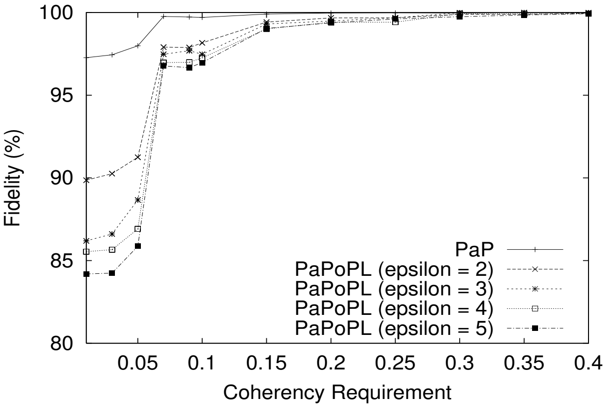

Figure 13: Effect of Epsilon on Fidelity

Our experimental results for different values of ϵ (for an MF of 16) are shown in Figure 13. Comparing the curves for PaP and PaPoPL for $\epsilon = 2$ shows that PaP offers better fidelity than PaPoPL over the entire range of tcr values. This is intutive as the fidelity offered by Pull is less than that offered by PaP. The server does not assign a lease to all the clients and some of the clients have to resort to Pull. These clients have less fidelity than PaP and hence the overall fidelity across all the clients is less than that offered by PaP. The figure also shows that just as with PaP, ϵ remains an important tunable parameter even for PaPoPL: With decreasing ϵ values, PaPoPL offers better performance in terms of fidelity.

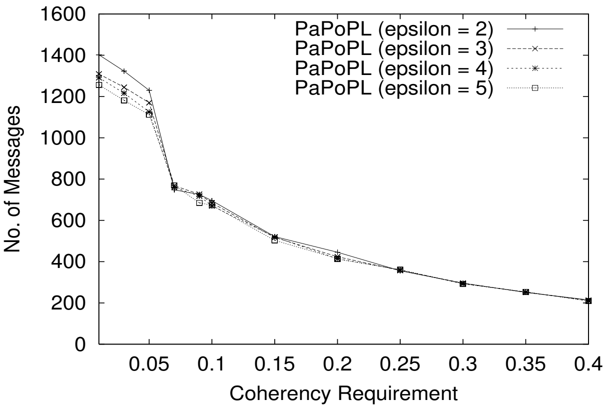

Figure 14: Effect of Epsilon on Message Overheads

Having examined the fidelity of PaPoPL, let us look at its communication overheads (Figure 14). It is observed from the graph that as the value of ϵ increases the number of messages goes on decreasing. The difference in the number of messages can be observed only for stringent tcr values. This reduction in the network overhead can be explained as follows : Suppose the expected pulling time of the proxy is 30. For an ϵ of 2, if the proxy does not pull at the expected time then the server waits till time 32 for the proxy to pull. In case ϵ is 5, the server would have waited till 35. Hence for an ϵ of 2, the server will push changes from time 33 to 35 which in case of ϵ of 5 will not be pushed by the server. Because of these extra pushes there is a difference in the number of messages for different values of ϵ. The probablity of a large change within a short time of 2-3 seconds is less and hence the difference in the number of messages is noticeble only for stringent tcr values.

We also studied the effect of changing value of MF on fidelity. The value of MF was varied from 4 to 64 and it was observed that the fidelity offered by PaPoPL decreased very slightly with increasing MF. When the MF is small the proxy has the ability to cater to the dynamics of the data in a better manner. A client that needs a PaP connection (and currently has Pull) will be assigned the correct connection earlier with smaller MF. This improvement in the fidelity is seen especially for strict temporal coherency values. The flip side to this increased fidelity is the slightly incresed message overhead. These results showed that as long as the MF value is not set very low, large differences are not likely when MF is varied.

Thus by adaptively choosing the right kind of connection (PaP/Pull) based on the currect trend in the data, PaPoPL offers better fidelity and overhead characteristics than PaPoP.

6 Summary of the Algorithms and Tuning their Parameters

In the previous sections, after analyzing the behavior of classical Pull and Push, we discussed two simple combinations of Push and Pull along with two enhancements to these combinations. Choosing an algorithm in practice requires an understanding of the importance of fidelity, scalability, and resiliency for a particular application. A starting point for this choice can be the following table summarizing the characteristics of the six algorithms. Except for Push which achieves 100% fidelity, all others have tunable parameters which allow us to tradeoff

Table 4: Behavioral Characteristics of Data Dissemination Algorithms

Algorithm	**Fidelity**	**Overheads (Scalability)**			**Resiliency**
		Communication	*Computation*	*State Space*	
Push	High	Low	High	High	Low
Pull	Low (for small tcrs) High (for large tcrs)	High	Low	Low	High
PaP	Adjustable (fine grain)	Low/Medium	Low/Medium	High	Graceful degradation
PoP	Adjustable (coarse grain)	Low/Medium	Low/Medium	Low/Medium	Delayed graceful degradation
PaPoP	Adjustable	Low/Medium	Low/Medium	Low/Medium	Graceful degradation
PaPoPL	Adjustable (fine grain)	Low/Medium (< PaPoP)	Medium	Medium	Graceful degradation

achieved fidelity for scalability and resiliency.

In the rest of this Section, we briefly outline some of the approaches available to effect this tuning, focusing on (a) the choice of ϵ used in PaP and (b) the choices made by PoP and PaPoP.

Using PaP, if the user wants high fidelity then we must set ϵ close to TTR_{min}.

The parameter ϵ of the PaP algorithm can be used to vary the network overhead offered by the algorithm as well as the fidelity. As seen earlier, there is a decrease in the average overhead as the value of ϵ increases.

We obtained linear equations relating the average overheads with ϵ and tcr by doing multiple regression analysis of the data. Averaging across all the stocks we obtained:

$$Overheads = A \times \epsilon + B \times tcr + C$$

where A = -0.022, B = -3.93, and C = 35.3829 is the intercept. The values of A and B were nearly the same for all the stock traces but the intercept value showed a large variation. Hence in order to achieve the required overheads we can vary the value of ϵ and the intercept. The fidelity cannot be measured by the proxy and hence the above equation is given in terms of overheads.

Suppose the network bandwidth allocated to a particular user is fixed, that is, the user specifies the maximum network overhead that can he can afford. The proxy can use this value to change the parameters (namely ϵ) depending on the dynamics of the data. To tune the value of ϵ the following algorithm can be used:

- Initially determine the value of ϵ using the equation derived by stastical analysis

$$\epsilon = \frac{Required\ Overheads - B \times tcr - C}{A}$$

- The ϵ calculated in the above step is used as an initial value. After each polling by the proxy the value of the intercept (C) and ϵ are recalculated. First the value of the intercept (C) is calculated based on the network overhead observed during the last polling.

$$C = Observed\ Overheads - A \times \epsilon + B \times tcr$$

  Using this value of C the value of ϵ is recalculated. This new value of ϵ needs to be conveyed to the server, and can be done during the next pull by the proxy.

We used this approach to work within specified communication overheads. The fidelity observed when ϵ was calculated by this approach, for a given tcr, was nearly the same as the fidelity observed by running the PaP algorithm with the optimal of ϵ. This signifies that the algorithm succeeds in making the optimal choice of the parameter ϵ.

To improve the performance even further, the proxy can keep track of the values of the coefficients A,B and C for all the stocks that it has seen so far. For each new client if the proxy has the values of the coefficients for the stock that the client is interested in, then they can be used, else the proxy can use the values obtained for the average case.

Turning to PoP (and PaPoP as well as PaPoPL) it is clear from the results plotted in Figures 7 and 9 and tabulated in Table 3 that the way in which clients are categorized as push, PaP, or pull clients affects fidelity and overheads. So the system must dynamically and intelligently allocate the available resources amongst push, PaP, and pull clients. For example, (a) when the system has to scale to accommodate more clients, it should preempt a few push or PaP clients (ideally, those which can tolerate some loss in fidelity) and give those resources to pull clients; (b) if the number of clients accessing the server is very small, a server can allocate maximum resources to push or PaP clients thus ensuring higher fidelity. Thus, by varying the allocation of resources, a server can either ensure high fidelity or greater scalability. The lease duration parameter of PaPoPL helps to periodically adjust the categorization based on the dynamics of the data and the system load and bandwidth.

Finally, we would like to note that we have begun to investigate novel systematic approaches to parameter tuning based on control-theoretic principles. We have developed an alternative to the heuristic approach outlined in Section 2.1 for determining the time-to-refresh (e.g., Time-To-Live or TTL). The control objective is to decide the time of pull so that a change of c is not missed. In [20], we discuss the design and performance of a proportional controller designed for deciding when to pull next. Using traces in the current paper, we compared the performance of this feedback control based algorithm with the approach outlined in Section 2.1 and initial results show that it is possible to achieve similar or even higher fidelity while incurring lower communication overheads. Encouraged by these results we are exploring whether the principles underlying this control theoretic formulation lend themselves to scientific approaches for tuning the parameters found in the other dissemination algorithms proposed in the current paper. This seems plausible as there exist techniques in control theory to handle multi-variable systems. Since purely heuristic algorithms with several parameters are harder to tune, this is an attractive line for further investigation.

7 Related Work

Several research efforts have investigated the design of push-based and pull-based dissemination protocols from the server to the proxy, on the one hand, and the proxy to the client, on the other. Push-based techniques that have been recently developed include broadcast disks [1], continuous media streaming [3], publish/subscribe applications [19, 4], web-based push caching [14], and speculative dissemination [5]. Research on pull-based techniques has spanned the areas of web proxy caching and collaborative applications [7, 8, 22]. Whereas each of these efforts has focused on a particular dissemination protocol, few have focused on supporting multiple dissemination protocols in web environment.

Netscape has recently added push and pull capabilities to its Navigator browser specifically for dynamic documents [21]. Netscape Navigator 1.1 gives two new open standards-based mechanisms for handling dynamic documents. The mechanisms are (a) *Server push*, where the server sends a chunk of data; the browser displays the data but leaves the connection open; whenever the server desires, it continues to send more data and the browser displays it, leaving the connection open; and (b) *Client pull* where the server sends a chunk of data, including a directive (in the HTTP response or the document header) that says "reload this data in 5 seconds", or "go load this other URL in 10 seconds". After the specified amount of time has elapsed, the client does what it was told – either reloading the current data or getting new data. In server push, a HTTP connection is held open for an indefinite period of time (until the server knows it is done sending data to the client and sends a terminator, or until the client interrupts the connection). In contrast, in client pull, HTTP connections are never held open; rather, the client is told when to open a new connection, and what data to fetch when it does so. Server push is accomplished by using a variant of the MIME message format "multipart/mixed", which lets a single message (or HTTP response) contain many data items. Client pull is accomplished by an HTTP response header (or equivalent HTML tag) that tells the client what to do after some specified time delay. The computation, state space and bandwidth requirements in case of Server push will be at least as much as was discussed in section 2.4. In addition, since we are using HTTP MIME messages, the message overhead will be more (on average MIME messages are bigger than raw data). Because of the same reason, it is not feasible to use this scheme for highly dynamic data, where the changes are small and occur very rapidly. Also, it would be very difficult to funnel multiple connections into one connection as envisaged in our model 1 (see equation 3). This will clearly increase the space and computation requirements at the server. For the Client pull case, for reasons discussed in Section 7, it is very difficult to use this approach for highly dynamic data. Still, these mechanisms may be useful for *implementing* the algorithms discussed in this paper as they are supported by standard browsers.

Turning to the caching of dynamic data, techniques discussed in [16] primarily use push-based invalidation and employ dependence graphs to track the dependence between cached objects to determine which invalidations to push to a proxy and when. In contrast, we have looked at the problem of disseminating individual time-varying objects from servers to proxies.

Several research groups and startup companies have designed adaptive techniques for web workloads [2, 7, 13]. Whereas these efforts focus on reacting to network loads and/or failures as well dynamic routing of requests to nearby proxies, our effort focuses on adapting the dissemination protocol to changing system conditions.

The design of coherency mechanisms for web workloads has also received significant attention recently. Proposed techniques include strong and weak consistency [17] and the leases approach [10, 23]. Our contribution in this area lie in the definition of temporal coherency in combination with the fidelity requirements of users.

Finally, work on scalable and available replicated servers [24] and distributed servers [9] are also related to our goals. Whereas [24] addresses the issue of adaptively varying the consistency requirement in replicated servers based on network load and application-specific requirements, we focus on adapting the dissemination protocol for time-varying data.

We end this section with a detailed comparison of an alternative to our client-based prediction,, namely Server-based prediction for setting Time-to-Live attributes for Web objects [11, 15, 17]. PaP and PoP are based on using prediction capabilities at the clients/proxies. An alternative of course is to leave the prediction to the server. Such schemes are discussed in [11, 15, 17]. They use the `if-modified-since` field associated with the HTTP GET method (also known as conditional GET), together with the TTL (time-to-live) fields used in many of proxy caches. These schemes in general work as follows:

- Client does not use any TTR or prediction algorithm, but instead depends on some meta information associated with the data to decide the time at which to refresh the data.
- Since the server has access to all the data, it can use a prediction algorithm to predict a time when the data is going to change by tcr. The server then attaches this time value with outgoing data. Client will use this meta information to decide when to poll next. There is no need for a push connection.
- Since server has better access to data than client, server predictions will be in general more "accurate" than using a TTR algorithm at the client.

Though the *Server-Prediction* approach looks like a better option than PaP, it runs into following problems:

- the approach requires that previous history for the relevant data be maintained at the server. This will imply increased state information and computational needs at the server and will consequently adversely affect the scalability. Since in PoP (section 4) we reserve the pull method to serve clients when faced with problems of scalability, we prefer to make Pull relatively "light weight".
- the approach is more suitable for data that changes less frequently (e.g., say once every few hours). We are interested in web data that is highly dynamic and inherently unpredictable (e.g., data that changes every few seconds/minutes such as stock quotes). For dynamic data, the performance will be slightly better than Adaptive TTR, but at a cost of server resources and scalability.
- if the server prediction is wrong and still a change of interest occurs in data, the server is helpless since it cannot push the change to the client. The change is lost. This will never happen in PaP.

In summary, so far dynamic data has been handled at the server end [16, 11]; our approaches are motivated by the goal of offloading this work to proxies.

8 Concluding Remarks

The goal of our research project is the development of intelligent and adaptative dissemination algorithms for dynamic data. 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. To address this limitation, we proposed two combinations of push- and pull-based approaches to adaptively determine which approach is best suited at a particular instant. Our first technique (*PaP*) is inherently pull-based and maintains soft state at the server. The proxy is primarily responsible for pulling those changes that are of interest; the server, by virtue of its soft state, may optionally push additional updates the proxy, especially when there is a sudden surge in the rate of change that is yet to be detected by the proxy. Since the server maintains only soft state, it is neither required to push such updates to the proxy, nor does it need to recover this state in case of a failure. Our second technique (*PoP*) allows a server to adaptively choose between a push- or pull-based approach on a per-connection basis (depending on observed rate of change of the data item or the coherency requirements). We also showed how PoP can be extended to use PaP for some of its connections, leading to the algorithm PaPoP. This algorithm was further extended to PaPoPL where the PaP connections were leased and each PaP client was possibly recategorized upon the expiry of its lease.

A contribution of our work is the design of algorithms that allow a proxy or a server to efficiently determine *when* to switch from a pull-based approach to push or PaP and vice versa. These decisions are made based on (i) a client's temporal coherency requirements (tcr), (ii) characteristics of the data item, and (iii) capabilities of servers and proxies (e.g., a pure HTTP-based server precludes the use of push-based dissemination and necessitates the use of a pull-based approach by a proxy).

Our techniques are designed with the view of being *user-cognizant* (i.e., aware of user and application requirements), *intelligent* (i.e., have the ability to dynamically choose the most efficient set of mechanisms to service each application), and *adaptive* (i.e., have the ability to adapt a particular mechanism to changing network and workload characteristics). Our experimental results demonstrated that such tailored data dissemination is essential to meet diverse temporal coherency requirements, to be resilient to failures, and for the efficient and scalable utilization of server and network resources. To carry this to its logical next step, we briefly outlined a set of techniques, including using control theory, to tune the parameters used in the algorithms to meet a diverse set of requirements.

Currently we are extending the algorithms developed in this paper to design algorithms suitable for cooperative proxies and also for disseminating the results of continual queries [18, 6] posed over dynamic data.

References

[1] S. Acharya, M. J. Franklin and S. B. Zdonik, Balancing Push and Pull for Data Broadcast, *Procs. of the ACM SIGMOD Conference, May 1997*.

[2] FreeFlow Product Details, Akamai Inc., *http://www.akamai.com/service/freeflow.html, 1999*.

[3] E. Amir, S. McCanne and R. Katz, An Active Service Framework and its Application Real-time Multimedia Transcoding., *Proceedings of the ACM SIGCOM Conference, pages 178-189, September 1998*.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman An Efficient Multicast Protocol for Content-based Publish-Subscribe Systems, *International Conference on Distributed Computing Systems* 1999.

[5] A. Bestavros, Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time in Distributed Information Systems., *International Conference on Data Engineering, March 1996*.

[6] M. Bhide, K. Ramamritham, and P. Shenoy, Efficiently Maintaining Stock Portfoilis up-to-date on the Web,IEEE Research Issues in Data Engineering (RIDE'02) workshop, March 2002.

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

[8] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz and K. J. Worrel A Hierarchical Internet Object Cache., *Proceedings of the 1996 USENIX Technical Conference, January 1996*.

[9] D. M. Dias, W. Kish, R. Mukherjee and R. Tewari, A Scalable and Highly Available Server., *Proceedings of the IEEE Computer Conference (COMPCON), March 1996*.

[10] V. Duvvuri, P. Shenoy and R. Tewari, Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. *InfoCom* March 2000.

[11] V. Cate, Alex - A Global File System. *Proceedings of the 1992 USENIX File System Workshop*, pages 1-12, May 1992.

[12] 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*.

[13] A. Fox, Y. Chawate, S. D. Gribble and E. A. Brewer, Adapting to Network and Client Variations Using Active Proxies: Lessons and Perspectives., *IEEE Personal Communications, August 1998*.

[14] J. Gwertzman, M. Seltzer, The Case for Geographical Push Caching, *Proceedings of the 5th Annual Workshop on Hot Operating Systems, pages 51-55, May 1995*.

[15] J. Gwertzman, M. Seltzer, World-wide Web Cache Consistency, *Proceedings of 1996 USENIX Technical Conference, January 1996*.

[16] 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*.

[17] 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*.

[18] L. Liu, C. Pu and W. Tang, Continual Queries for Internet Scale Event-Driven Information Delivery, *IEEE Trans. on Knowledge and Data Engg., July/August 1999*.

[19] G. R. Malan, F. Jahanian and S. Subramanian, Salamander: A Push Based Distribution Substrate for Internet Applications., *Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997*.

[20] F. Marolia, K. M. Moudgalya and K. Ramamritham, When to Pull Dynamic Web Data? An Answer Using Classical Feedback Control (available from www.cse.iitb.ac.in/ krithi/papers/control_pull.ps.gz), 2002.

[21] An Exploration of Dynamic Documents, Netscape Inc., *http://home.netscape.com/assist/net_sites/pushpull.html*.

[22] Raghav Srinivasan, Chao Liang and Krithi Ramamritham, Maintaining Temporal Coherency of Virtual Data Warehouses, *The 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 2-4, 1998.

[23] J. Yin, L. Alvisi, M. Dahlin and C. Lin, Hierarchical Cache consistency in a WAN, *Proceedings of the USENIX Symposium on Internet Technologies and Systems, October 1999*.

[24] H. Yu and A. Vahdat, Design and Evaluation of a Continuous Consistency Model for Replicated Services, *Proceedings of OSDI*, October 2000.