An Efficient and Resilient Approach to Filtering and Disseminating Streaming Data

Shetal Shah   Shyamshankar Dharmarajan   Krithi Ramamritham

TCS Lab for Internet Research,
Dept of Comp Science and Engg,
Indian Institute of Technology Bombay,
Mumbai, India 400076.

Abstract

Many web users monitor dynamic data such as stock prices, real-time sensor data and traffic data for making on-line decisions. Instances of such data can be viewed as data streams. In this paper, we consider techniques for creating a resilient and efficient content distribution network for such dynamically changing streaming data. We address the problem of maintaining the coherency of dynamic data items in a network of repositories: data disseminated to one repository is filtered by that repository and disseminated to repositories dependent on it. Our method is resilient to link failures and repository failures. This resiliency implies that data fidelity is not lost even when the repository from which (or a communication path through which) a user obtains data experiences failures. Experimental evaluation, using real world traces of streaming data, demonstrates that (i) the (computational and communication) cost of adding this redundancy is low, and (ii) *surprisingly*, in many cases, adding resiliency enhancing features actually improves the fidelity provided by the system even in cases when there are no failures. To further enhance fidelity, we also propose efficient techniques for filtering data arriving at one repository and for scheduling the dissemination of filtered data to another repository. Our results show that the combination of resiliency enhancing and efficiency improving techniques in fact help derive the potential that push based systems are said to have in delivering 100% fidelity. Without them, computational and communication delays inherent in dissemination networks can lead to a large fidelity loss even in push based dissemination.

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

1 Introduction

The internet and the web are increasingly used to disseminate fast changing data like data collected from sensors, traffic and weather information, stock prices, sports scores, and even health monitoring information (http://www.openclinical.org/aispi_neoganesh.html). The data under consideration is *highly dynamic*, i.e., the data changes continuously and at a fast rate, *streaming*, i.e., new data can be viewed as being appended to the old or historical data, and *aperiodic*, i.e., the time between the updates and the value of the updates are not known apriori. Increasingly, users are interested in not only monitoring streaming data but in also using it for on-line decision making. The growth of the internet has made the problem of managing streaming, i.e., dynamic data both interesting and challenging.

Resource limitations at a source of dynamic data will limit the number of users that can be served directly by the source. A natural solution to this is to have a set of repositories which replicate the source data and serve the data to geographically closer users. Services like *Akamai.net* and IBM's edge server technology are exemplars of such networks of repositories, which aim to provide better services by shifting most of the work to the edge of the network (closer to the end users). But, although such systems scale quite well, if the data is changing at a fast rate, the quality of service at a repository farther from the data source would deteriorate. In general, replication can reduce the load on the sources, but replication of time-varying data introduces new challenges. First, data at the repositories needs to be coherent with the source. Second, unless updates to the data are carefully disseminated from sources to repositories, the communication and computation overheads involved in such dissemination can themselves result in delays as well as scalability problems, further contributing to loss of data coherency.

In situations where the data is to be used for on-line decision making, users specify the bound on the tolerable imprecision associated with each requested data item. This can be viewed as the *coherency requirement (cr)* associated with the data. The coherency requirement associated with a time-varying data item depends on the nature of the item and user tolerances. For example, a user involved in exploiting exchange disparities in different markets or an online stock trader may impose stringent coherency requirements (e.g., the stock price should never be out-of-sync by more than one cent from the actual value) whereas a casual observer of currency exchange rate fluctuations or stock prices may be content with a less stringent coherency requirement. The basic framework underlying the coherency model is outlined in Section 2.

The focus of our work is to design and build a dynamic data distribution system which is *coherency-preserving*, i.e., the delivered data must preserve associated coherency requirements, *resilient* to failures, and *efficient*, i.e., the system should be able to provide high fidelity even with a large number of users and data. We consider a system in which the necessary changes are *push*ed to the users i.e., users are automatically informed about changes of interest, rather than each user independently polling the source(s) for changes of interest. The efficacy of the system's response to users' requests can then be evaluated with respect to the offered (a) *Fidelity* which is indicative of the degree to which the user's coherency needs are satisfied, and (b) *Number of Messages*, measured in terms of the number of messages that are exchanged over the network to provide this fidelity. Unless designed carefully, contrary to folklore, even push-based systems experience considerable loss in fidelity due to message delays and processing costs.

The contribution of this paper lies in efficient solutions to the three major problems that need to be solved to effectively address this challenge.

1) Construction of an effective dissemination network of repositories. This network of repositories is a logical overlay network constructed taking into account the data and coherency needs of users attached to each repository, the expected delays at each repository as well as delays along the communication paths connecting the repositories. We call this network a *dynamic data dissemination graph*, (d^3g).

In [26] we presented $LeLA$, an algorithm to build a d^3g. However, further experimentation showed $LeLA$ to be unable to cope when a large number of data items were being disseminated, with or without failures in the network. This motivated us to investigate techniques to build dissemination networks that are scalable and resilient. The new algorithm $DiTA$, presented in Section 3, for the construction of dissemination graph has these features. In $DiTA$, repositories with more stringent coherency requirements are placed closer to the source in the network as they are likely to get more updates than with looser coherency requirements. By placing them closer to the source, we can reduce the number of messages in the system and this in turn is likely to improve the fidelity of the system.

Since a repository will typically need multiple data items, each at different coherency requirements, in $DiTA$, we build a *dynamic data dissemination tree*, d^3t, for each data item. Hence, for some data items (for which it has stringent coherency requirements), the repository will be closer to the respective sources. When multiple data items are considered, each physical repository can be seen as cooperating with other repositories in the physical network, forming a peer-to-peer relationship. Our performance study, using real world traces of real-time (stock) streams, shows that *$DiTA$ delivers better fidelity than $LeLA$*: fidelity losses were often found to be almost an order of magnitude lower than that of $LeLA$.

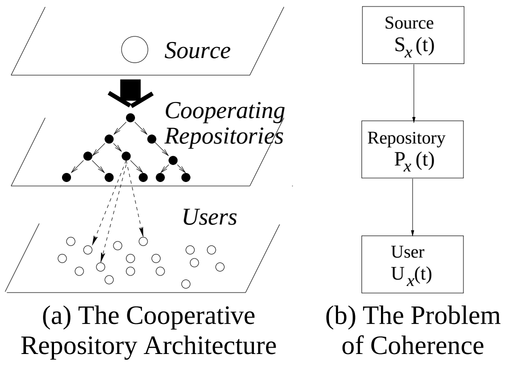
2) Provision for the dissemination of dynamic data in spite of failures in the overlay network. Our approach, presented in Section 4, to handle repository failures as well as the communication link failures is based on adding back-up parents to a dependent, but of significance is the design feature that a back-up parent is asked to deliver data with coherency that is less stringent than that associated with the parent. This reduces the overheads of providing resiliency, yet allows the dependent to determine if a parent has failed. If such a failure is detected, measures are provided to find an alternative parent. Performance studies show that our measures to add resiliency to the dissemination tree result in an interesting and useful side effect: In many cases, *contrary to expectations, when no failures are present, fidelity in fact improves over the no-resiliency case.* In Section 4 we explain the reason for this surprising result.

3) Efficient filtering and scheduling techniques for repositories. In a typical d^3t, a source or a repository receives updates for data items and selectively disseminates them to its downstream repositories or to users directly connected to it. Given the list of requests associated with a repository, for every update to a data item, the repository essentially identifies the requests that match this update. To achieve high fidelity we must reduce the delay (a) in performing this matching and (b) in pushing the changes to the interested dependents.

Our approach presented in Section 5, makes use of the observation that *it is not always necessary to disseminate the exact values of the most recent updates, as long as the values presented preserve the coherency of the data. Disseminating a* pseudo-value *might be beneficial especially when the data value oscillates around the pseudo-value.* Furthermore, simple yet effective matching techniques are incorporated within a repository to make data dissemination through a repository highly efficient. Performance results show that such informed filtering and dissemination of updates leads to better fidelity. Often an order of magnitude reduction is observed in the fidelity losses that are inevitable due to dissemination delays.

In summary, this paper presents an efficient method for constructing an overlay network for dynamic data dissemination. In addition, it presents resiliency and efficiency enhancement techniques that allow the handling of failures and permit the dissemination to scale to a large number of users and data items. We discuss related work in Section 6 and then conclude the paper.

2 The Basic Framework: Data Coherency and Overlay Network



(a) The Cooperative Repository Architecture (b) The Problem of Coherence

Figure 1: The Basic Framework

As shown in Figure 1(a), we build a network of sources and repositories with users connecting to the repositories, and repositories deriving their data needs from users' data and coherency requirements. Suppose a coherency requirement (c) is associated with a data item, to denote the maximum permissible deviation of the user's view from the value of data d at the source. Generally, 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., weather information where the temperature value should never be out-of-sync by more than two degrees). In this paper, we only consider coherence requirements specified in terms of the value of the object; maintaining coherence requirements in units of time is a simpler problem that requires less sophisticated techniques (e.g., push every 5 minutes). Each data item in the repository from which a user obtains data must be refreshed in such a way that the user-specified coherency requirements are maintained. Formally, let $S_x(t)$ and $U_x(t)$ denote the value of a data item x at the source and the user, respectively, at time t (see Figure 1(b)). Then, to maintain coherency , we should have

$$|U_x(t) - S_x(t)| \leq c$$

. The issue of which repository a user should connect to is a separate problem and is not addressed in this paper. We assume that the repositories transmit the data updates to the users with negligible delays and hence we focus on maintaining coherence at the repositories, i.e.,

$$|P_x(t) - S_x(t)| \leq c$$

Empirically, fidelity f observed by a user can be defined to be the total length of time for which the above inequality holds, normalized by the total length of the observations. A good coherency mechanism must provide high fidelity at low cost. Note that although Figure 1(b) shows a single data repository, the coherency requirements are no different if there are multiple data repositories acting as intermediaries between the source and the end-user.

For each data item we build a logical overlay network or dynamic data dissemination tree, (d^3t), as described below. Consider a data item x. We assume that x is served by only one source. It is possible to extend the algorithm to deal with multiple sources, but for simplicity we do not consider this case here. Let repositories $R_1, \ldots, R_n$ be interested in x. The source directly serves some of these repositories. These repositories in turn serve a subset of the remaining repositories such that the resulting network is in the form a tree rooted at the source and consisting of repositories $R_1, \ldots, R_n$. The children of a node in the tree are also called the *dependents* of the node. Thus, a repository serves not only its users but also its dependent repositories. Since the repository disseminates updates to its users and dependents, the coherency requirement of a repository should be the most stringent requirement that it has to serve. When a data-change occurs at the source, it checks which of its direct and indirect dependents are interested in the change and *pushes* the change to them. (This check is enabled by a *Centralized (source-based)* component algorithm of $LeLA$ [26], and is also used by $DiTA$.) When a repository gets a data update, it in turn pushes the change to its dependents. Each repository acts as a filter where it sends only the updates of interest further down. Thus, the view of x at any repository is a projection of the changes taking place to x at the source.

3 Building a d^3t.

In this section, we show how to build a d^3t for a data item x, given the coherency requirements of each repository interested in x. The basic algorithm is described here and the next section shows how to make the d^3t resilient to failures.

Suppose we are given the physical layout of the communication network in the form of a graph, where the graph consists of a set of sources, repositories and the underlying network. In the sequel, when we refer to a d^3t we mean, a d^3t for x. The root of the d^3t is the source which serves x. A repository P serving repository Q with data x, is called the *parent* of Q for data item x and Q is called the *dependent* of P for x. The length of the path from a repository in the d^3t to the root, i.e., the source, is the *level* of the repository in the d^3t. In other words, it is the number of logical links between the source and the repository. The source is considered to be at level 0. The dependents of the source are at level 1.

We do not want to overload a repository and hence we place a limit on the number of unique <dependent, data item> pairs that it can serve. A repository should ideally serve at least as many unique pairs as the number of data items served to it. If a repository is currently serving less than this fixed number, then we say that the repository has the *resources* to serve a new dependent. Thus, $DiTA$ has a built-in limit on the resources that a repository offers towards cooperation.

A repository R interested in data item x requests the source for insertion. When the source gets the request it checks if it has enough resources to service R. If it has the resources or if the d^3t consists of only one node, i.e., the source, R is made a dependent of the source in the d^3t. If the source does not have the resources, as described next, it determines the most suitable subtree rooted at its dependents for the insertion of R.

Each repository P in a d^3t maintains the least stringent coherence requirement for that data item at each level in the subtree rooted at P. Every time a new node is inserted in the d^3t, we update the data-structures at all its ancestors if its coherence requirement is the least stringent in its level. This information is used by P to determine the most suitable subtree rooted at its dependents for the insertion of R. The subtree is chosen such that the level of R in the d^3t is the smallest possible and that communication delays between R and its parent are small. This is recursively applied to select subtrees in the subtree, till we reach a node Q such that

1. Q has data that is stringent enough to meet R's requirements and Q has the resources to serve R. In this case, R is made the dependent of Q.

2. Coherency requirement of Q is less stringent than R. In this case R pushes Q down in the subtree. It replaces Q. The parent of Q now serves R and R in turn serves Q. R also serves as many dependents of Q as it can.

The motivation behind this replacement technique is to get a d^3t where repositories with more stringent coherencies serve repositories with loose coherencies.

In the rest of the paper, we refer to the above algorithm as Data-$i$tem-at-a-Time-Algorithm ($DiTA$). $DiTA$ requires very little book-keeping and, experimental results, show that it indeed produces d^3ts that deliver data with high fidelity, and in fact is almost an order of magnitude better than $LeLA$ introduced in [26].

We now present the experimental methodology and then the results for the performance evaluation of the d^3 construction algorithms.

Traces – Collection procedure and characteristics: The performance characteristics of our solution are investigated 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) obtained by continuously polling *http://finance.yahoo.com*. We collected 1000 traces making sure that the corresponding stocks did see some trading during that day. The details of some of the traces are listed in the table below to suggest the characteristics of the traces used. (*Max* and *Min* refer to the maximum and minimum prices observed in the 10000 values polled during the indicated *Time Interval* on the given $Date$ in Jan/Feb 2002.) As we can see, we were able to obtain a new data value approximately once per second. Since stock prices change at a slower rate than once per second, the traces can be considered to be "real-time" traces.

Company	*Date*	*Time Interval*	*Min*	*Max*
Microsoft	Feb 12	22:46-01:46 hours	60.09	60.85
SUNW	Feb 1	21:30-01:22 hours	10.60	10.99
DELL	Jan 30	00:43-04:12 hours	27.16	28.26
QCOM	Feb 12	22:46-01:46 hours	40.38	41.23
INTC	Jan 30	00:43-04:12 hours	33.66	34.239
Oracle	Feb 1	21:30-01:22 hours	16.51	17.10

Characteristics of some of the Traces used for the experiment

Repositories – Data, Coherency and Cooperation characteristics: Each repository requests a subset of data items, with a particular data item chosen with 50% probability. A coherency requirement c is associated with each of the chosen data items. We use different mixes of data coherency. Specifically, the c's associated with data in a repository are a mix of stringent tolerances (varying from \$0.01 to 0.05) and less stringent tolerances (varying from \$0.5 to 0.99). T% of the data items have stringent coherency requirements at each repository (the remaining $(100 - T)$%, of data items have less stringent coherency requirements).

Physical Network – topology and delays: The physical network consists of nodes (routers and repositories) and links. The router topology was generated using BRITE (http://www.cs.bu.edu/brite). Once the router topology was generated we randomly placed the repositories and the sources in the same plane as that of the routers and connected each to the closest router. For each repository, data items of interest were first generated and then coherencies were chosen from the desired range.

Our experiments use node-node communication delays derived from a heavy tailed Pareto [24] distribution: $x \rightarrow \frac{1}{x^{\frac{1}{\alpha}}} + x_1$ where α is given by $\frac{\bar{x}}{\bar{x}-1}$, $\bar{x}$ being the mean and x_1 is the minimum delay a link can have. For our experiments, $\bar{x}$ was 15 ms (milli secs) and x_1 was 2 ms. As a result, the average nominal node-node delay in our networks was around 20-30 ms. This is lower than the delays reported based on measurements done on the internet [10]. We also present the results obtained at high link delays.

Unless otherwise specified, computational delay incurred at a repository to disseminate an update to a dependent is taken to be 12.5 ms. This includes the time to perform any checks to examine whether an update needs to be propagated to a dependent and the time to prepare an update for transmission to a dependent (details of these are given in Section 5). In the presence of complex query processing at repositories, for example, if a repository aggregates information before transmitting updates to its dependents, this processing time can be considerable and hence the above default value for the computational delay. We also measured the effect of other values of computational delays on fidelity.

Metrics: The key metric for our experiments is the loss in fidelity of the data. Recall that fidelity is the degree to which a user's coherency requirements are met and is measured as the total length of time for which the inequality $|P(t) - S(t)| \leq c$ holds (normalized by the total length of the observations). The fidelity of a repository is the mean fidelity over all data items stored at that repository, while the overall fidelity of the system is the mean fidelity of all repositories. The loss in fidelity is simply $(100\% - \text{fidelity})$. Clearly, the lower this value, the better the overall performance of a dissemination algorithm.

Another secondary metric that we use is *Number of Messages* in the system. This is the total number of data updates pushed. This gives us an indication of the load on the network and hence the possible effect on fidelity.

Performance Evaluation: In [26], we have already shown how filtering of updates at the repositories based on coherency requirements improves fidelity and that cooperation up to a certain point is beneficial.

For the base performance measurement, we used a network topology consisting of 600 routers, 100 repositories and 4 servers. The number of data items that a server was servicing was varied from 25 to 250, i.e., the total number of data items served by all the servers was varied from 100 to 1000 (corresponding, say, to the most traded 1000 stocks in a market). Also, T the parameter that adjusts the data coherency mix, was varied from 20 to 80. $LeLA$ served as a benchmark against which to compare $DiTA$. The results presented are averaged over at least 5 different set of traces.

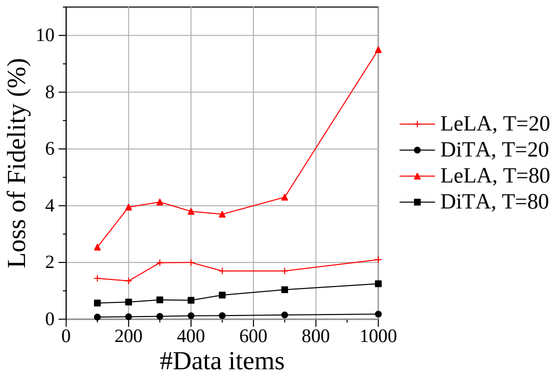


Figure 2: Performance of $DiTA$ versus $LeLA$ for Different Number of Data Items

We can clearly see from Figure 2 that $DiTA$ does much better than $LeLA$. (The small apparent reduction in fidelity loss values as we go from 300 to 500 can be attributed to the small differences in the d^3t's constructed by $LeLA$ for different number of data items). Specifically, for T=80%, whereas $DiTA$ has between 0.5 and 1.25% loss of fidelity, $LeLA$ has between 2.5 and 9.5% loss. For T=20%, the loss for $DiTA$ is an order of magnitude lower than that of $LeLA$. We noticed that in $DiTA$ a repository on an average served lesser number of unique <dependent, data item pairs> than $LeLA$. This amounts to less work done at a node in $DiTA$ and this is a primary contributor to $DiTA$'s superior performance.

Each node in $DiTA$ does less work than its counterpart in $LeLA$. As a result, the height of the dissemination tree in $DiTA$ can be expected to be more than that in $LeLA$. Thus, only when computation delays at a node per update are very low, or link delays are large, can $LeLA$ expect to have an edge. To test this hypothesis, the link delays were varied from 12.5 ms to 110ms. The computation delays were varied from 0.5 ms to 12.5 ms. We found that the behaviour of $LeLA$ was better (by just 0.5%) than that of $DiTA$ only for very high link delays (110ms) and for negligible computational delays (0.5 ms). For all other delays, $DiTA$ does substantially better than $LeLA$ (difference in fidelity is 1-3%). At high link delays, $LeLA$ had a maximum height of 4 whereas $DiTA$ had a height of 6 on an average and a maximum height of 10. Fortunately, these high link delays are not very common on the internet [10] and hence in practice $DiTA$ is preferable. Another interesting observation was that at high link delays, the fidelity offered by $DiTA$ did not change much with change in computational delays. This is another indication of the dominance of link delays on the performance of $DiTA$.

4 Enhancing the Resiliency of the Repository Network

Two classical approaches for fault tolerance are to have either active backups or passive backups. The latter takes less time to deal with a failure but increases the normal load on the system. In our case, the increased load can in turn lead to loss of fidelity. Clearly, it will be better to achieve a sound compromise: fidelity loss incurred upon failure should be low, but the fault-tolerance mechanism should not degrade normal operation. We achieve this compromise by using active backup-parents, but so that the resulting overheads do not lead to loss of fidelity during normal operations, the backup-parent serves data to a dependent Q with a coherency $c_B > c$.

Once we fix c_B we can calculate the expected number of updates lost by Q in case of a failure assuming that data changes as a random walk on a line. (If all changes are less than c_B then we will not know when parent P fails. A possible embellishment to address this is to make P send periodic "I'm alive" messages.) Once P fails, Q requests B to serve it the data at c. When P recovers from the failure, Q requests B to serve the data item at c_B.

Note that this simple approach continues to provide data, albeit with a lower coherency, to a dependent even when a parent fails. Note that even if all parents serving a repository fail, this will not disrupt the data dissemination. But if a back-up parent also fails, then the repository will not get the data item(s) till one of them resumes service. In short, a back-up parent is not backed up. We now elaborate upon the choice B and c_B.

4.1 Choice of c_B Using a Probabilistic Model

For the sake of simplicity we set c_B, the coherency maintained by the backup-parent as a multiple of c, i.e., $c_B = k * c$. Choice of k is important as it will decide how many updates will be missed by the dependent on average in case P fails. If k is small, more particularly, if $k = 1$, then both the parent and the back-up parent will send all the updates to the dependent and we will incur high computational and communication overheads. If k is set at a high value we might miss a large number of changes. So, k depends on the acceptable overheads imposed on the backup parent and the acceptable number of missed updates. To calculate the number of missed updates, we have two options: (1) observe from sample runs, or (2) develop an analytical model. We choose the latter since it gives us a value that will hold independent of the dynamics of the data.

To simplify the treatment, we assume that the data values change up or down with uniform probability, i.e., the probability of an increase in data value is same as that of a decrease. No assumptions are made about the unit of change or the time taken for a change. Using a Markov Chain model (the detailed analysis is given in [27]) we calculated $\#Misses = 2k^2 - 2$.

($2k^2 - 2$) indicates the number of updates a dependent will miss, on an average, before it detects that there is a failure and hence is not getting updates from its parent. Depending on the number of updates a dependent may be willing to miss, we can set the value of k. For $k = 2$, $\#Misses = 8 - 2 = 6$. For dynamic data that does not exhibit uniform change the expected number of misses may vary from that calculated above. Further, even $(2k^2 - 2)$ is likely to be pessimistic. We calculated the number of updates that a repository got from the real parent for two consecutive updates from the backup-parent. This was averaged over 100 repositories and 100 different traces and we found that the actual misses were at most $2k^2 - 2$. As k increases, the number of actual misses are quite less in comparison to the expected number of misses. (For $k = 4, \#actual\ misses = 17$, $\#expected\, misses = 30$.)

Given the small number of missed updates for $k = 2$, we chose this value of k in our experimentation.

4.2 Choice of back-up parents

Let Q be a repository that wants a back-up parent for data item x. Let P be the parent of Q in the d^3t for x.

Consider the siblings of P. If P does not have any siblings then consider the siblings of the first nearest ancestor of P with a sibling. One of the siblings is randomly chosen to be the back-up parent of Q. Let this repository be B. In case the coherency at which Q wants x from B is less then the coherency at which B wants x, the parent of B is asked to serve x to B with the required tighter coherency. Note that the coherency increase will be at one level only: since the parent of B is also an ancestor of Q, it will be receiving updates of x at least at the coherency requirement of Q.

An advantage of choosing a sibling, as opposed to any other repository in the tree, is that the change in coherency requirement is not percolated all the way to the source. However, choosing a sibling might not be advantageous all the time. If an ancestor of P and B is heavily loaded than the delay due to the load will be reflected in the updates of both the B and P. This might result in additional loss in fidelity. Note that in case the d^3t is a skinny tree of repositories, then the source might finally become the back-up parent of Q for x.

4.3 Effect of Repository failures on Loss of Fidelity

The kind of failures we are looking at are memory less infrequent failures. So we use an exponential probability distribution $Pr(X > t) = e^{-\lambda t}$ to generate both the time to failure and the time to recover. Since the failures are infrequent we use a very small value of $\lambda = \lambda_1$ to generate the time to failure. We model transient failures (i.e. fast recovery) using a large value of $\lambda = \lambda_2$, ($\lambda_2 > 1$), to calculate the time taken for recovery. As a corollary, failures that require restart of the repositories (i.e. slow recovery) is modeled using a very small value of λ_2, ($\lambda_2 < 1$).

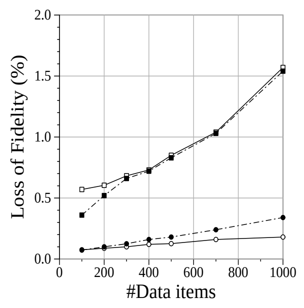
Between link failures and repository failures, the one that affects fidelity more is repository failure since a repository failure affects all its dependents whereas a link failure directly will typically affect only the dependents connected to that link. A link failure can be modeled as a partial failure of a repository - wherein for only some of its dependents the repository has failed but for others it has not. We have, however, not modeled such cases.

We would like to mention here that though our failure model is incomplete (i.e. link failures are not explicitly modeled) our solutions are complete, they will work in the presence of both repository failures and link failures.

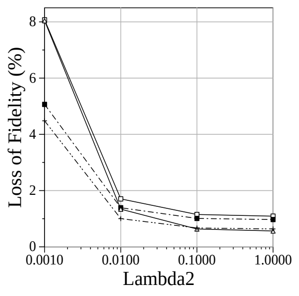
4.4 Performance Evaluation

Figure 3(a) shows the effect of adding resiliency. Back-up parents were calculated as mentioned in Section 4.2. The value of k was set to 2. We see that, where fidelity loss increases due to resiliency, the increase is small, less than 0.5%. On the other hand, in some cases, e.g., for small number of data items we observe that the resulting fidelity actually improves because of resiliency!

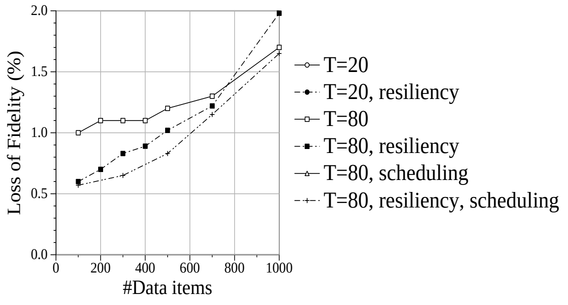
We wanted to understand the behavior in Figure 3(a) better. To this end we examined the number of updates disseminated by DiTA with and without resiliency. We saw an increase in the updates disseminated (about 60%) in the network due to resiliency (in the absence of failures). In spite of this we see that the fidelity offered by



(a) In the Absence of Failures



(b) Varying λ_2 (Recovery Times), for 100 Data items



(c) Under Quick Recovery, $\lambda_2 = 2$

Figure 3: Effect of Resiliency on Fidelity

the system actually improves! For 100 data items we observed that 23% of the updates sent by back-up parents were actually further disseminated by the dependents. Some of the updates sent by the back-up parent reached the dependents before the updates the parent sent (the back-up parent was less loaded than the parent) and in some cases the values sent by the back-up parent were different from that of the parent (since they see different views due to different coherency requirements). This leads to both the increase in the number of updates disseminated and also in the decrease in the loss in fidelity. However, when the back-up parents are loaded, the updates sent by them will typically reach later than that sent by the parents. (Back-up dependents are processed after the real dependents at any repository). Here the work done by the back-up parent is of no use to the dependent. This increases the loss in fidelity. The dependent tries to control the loss by discarding updates with time-stamps earlier than what it currently has for the same data item.

Finally we examine the performance under failures. For value of $\lambda_2 = 0.001$, $\frac{2}{3}$ of the failures were less than 500 sec, most of these were less than 200 ms. The remaining $\frac{1}{3}$ of the failures were such that most of the values were greater than 700 ms, the maximum being 1400 ms. We choose the value of λ_1 as 0.0001 to get a non-trivial number of failures in the system. During the experiment, About 80-90% of the repositories experienced at least one failure, and the maximum number of failures in the system at any given time for $\lambda_2 = 0.001$ was around 12. For $\lambda_2 = 0.01$, the maximum number of failures was 5 and for $\lambda_2 = 0.1$, the maximum failures was 2. These are admittedly indicative of pessimistic failure situations but we wanted to stress test our algorithms to show that they deliver good fidelity even under under high failure rates. Figure 3(b) shows that, as expected, adding resiliency improves fidelity in failure situations. The graphs that include the behavior of "scheduling" will be explained later in Section 5.

Figure 3(c) shows the effect of quick recovery in the network. The value of λ_1 was 0.0001 and that of λ_2 was 2. The average recovery time was less than a second. For high coherence requirements, resiliency improves fidelity even for transient failures. However, with resiliency we notice that with a very large number of data items, for e.g., 1000, fidelity drops even though it is less than 1%. This is because, at this point, the cost of resiliency exceeds the benefits obtained by it, i.e., the updates sent by the back up parent reach the dependent after those sent by the parent and hence this increases the lost in fidelity.

5 Reducing the Delay at a Repository

In this section we focus our attention on the development and evaluation of techniques used to reduce the fidelity losses implied by

- **Queuing delays:** Whenever an update arrives, it is added to an update queue. The time delay between the arrival of the update and the time when its processing is started, is termed as the queuing delay.
- **Processing delays:** When a data update reaches the head of the queue, it is taken up for processing. Dependents and their data coherency requirements are checked to decide if the update should be processed (check delay) and pushed to a specific dependent (computation delay is the delay associated with computing the data to be pushed and actually pushing it).

Our aim is to improve the average fidelity over all the repositories, which implies reducing the average delay between a data update and the time at which each interested repository receives the update. This is done by (a) *better filtering of updates, i.e., reducing the processing delay in determining if an update needs to be disseminated to one or more dependents* and (b) *better scheduling of the disseminations*. The details of how these are achieved is discussed in the next two subsections.

5.1 Better Filtering of Updates

For each dependent, a repository maintains the coherency requirement, cr, and the value pushed last for that dependent. Any new update must be pushed to a dependent if the new value changes by cr. Thus associated with each dependent is an upper bound u_b =

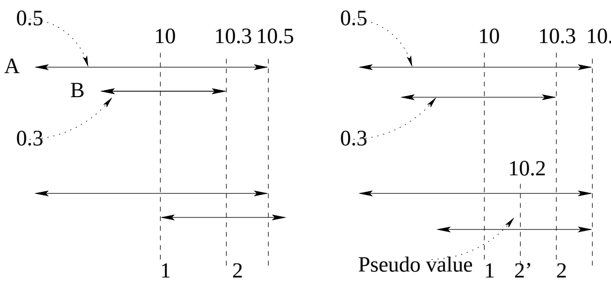


Figure 4: (left) Consider 2 dependents needing the same data item: A with cr = 0.5 and B with 0.3. Let the last values pushed to these dependents be 10. Then, B's window is (9.7, 10.3). When update 10.3 arrives it has to be pushed only to B, making the window for B (10, 10.6). A subsequent update of 10.55 will have to be sent only to A. (right) Changed scenario when a pseudo value of 10.2 is pushed to B.

$last\ pushed\ value\ +\ cr$ and a lower bound $l_b = last\ pushed\ value\ -\ cr$. Until a new value is pushed the user of the data knows that the data value lies in the window (lower bound, upper bound).

It is important that a repository that receives an update efficiently checks if the update lies outside a window. To reduce the number of checks, it is preferable to order the dependents' needs by some parameter. We use cr as the ordering parameter, so that when a new update arrives, the ordered list of dependents can be searched to determine the largest cr that demands a dissemination. All the dependents with cr less than this largest cr must also be notified. As it turns out, using this ordering is not as straightforward as described above because the fact that a dependent with a lower cr does not require a push does not mean that one with a higher cr also does not need the data. These are illustrated in Figure 4(left). In fact, we can have dependents with same cr, wherein for a given update, one of the dependents requires a dissemination, while the other does not. To address these problems, we impose some restrictions on the ordered dependent list, so that the following can be made to hold: (1) if an update has to be pushed to a dependent with cr c, then it needs to be pushed to dependents with coherency c', where $c' \leq c$. (2) More importantly, if a dependent with cr c does not require a push, no dependent with cr c', such that $c' > c$ will require a push. The restrictions that help us achieve the above are:

1. All dependents with the same cr have the same view of the data.

2. We consider each dependent as a coherency window than just a cr value. If v is the view of the data value for a dependent with cr c, the coherency window is the set of values lying between $(v - c)$ and $(v + c)$. We restrict the value sent to a dependent such that the coherency window for the dependent with larger cr bounds the coherency window for the smaller cr, i.e., for dependent r_1 with bounds $l_1 = (v_1 - c_1)$ and $u_1 = (v_1 + c_1)$ and r_2 with window $l_2 = (v_2 - c_2)$ to $u_2 = (v_2 + c_2)$,
$$c_1 < c_2 \quad \Rightarrow \quad l_2 \leq l_1 \quad and \quad u_2 \geq u_1 \qquad (1)$$

Consider the same scenario as described by Figure 4(left). Initially, the coherency windows for the dependents A and B are (9.5,10.5) and (9.7,10.3). When update of 10.3 arrives, if the value of 10.3 is pushed to client B, the new coherency window for it will be 10.0-10.6 which goes beyond the window of a dependent with cr 0.5. This violates our requirements above. Therefore in this case, instead of 10.3, a pseudo value of 10.2, which is closest to the actual value and satisfies both the query constraint and the bound condition is pushed to client B. Figure 4(right) explains this pictorially.

The rationale for the choice is that the view of the data value seen by the dependents need not be the actual value, but can even be a coherency window of width 2 cr about a *pseudo value*, such that the actual value is guaranteed to lie within that window and it satisfies the restrictions described above.

Thus, when a new update arrives, a search of the ordered list creates a threshold such that all dependents with cr less than the threshold require the update to be pushed. If r_b is the *bounding dependent* i.e., the dependent with the smallest cr larger than threshold and (l_b, u_b) are the lower and upper bounds of r_b, for each of these dependents with cr of c_i, the *pseudo value* to be pushed is computed using the function

$$Bound(v, c_i, r_b) = \begin{cases} (l_b + c_i) & \text{if } v < l_b + c_i \\ (u_b - c_i) & \text{if } v > u_b - c_i \\ v & \text{otherwise} \end{cases}$$

where v is the actual update value. As shown in Section 5.3, this optimization, referred to as *dependent ordering*, substantially reduces the number of pushes and also the loss of fidelity. To enable *dependent ordering* to facilitate fast insertions, deletions and searches, but allow ordered retrieval, we use R-B trees which allow log(n) insertions, deletions and searches.

5.2 Scheduling for performance improvement

This section establishes the criteria that should be used for determining the order in which (i) the updates must be processed and (ii) an update should be propagated to its dependents so that the overall fidelity across all repositories is maximized. Consider a set of updates, $u_1, u_2, \ldots u_n$, waiting in the update queue to be processed. Let this be the order for processing the updates as well. Let $C(u_1), C(u_2) \ldots C(u_n)$ be the time delay for processing these updates ($cost$) and $b(u_1), b(u_2) \ldots b(u_n)$ be the total number of descendants that would be benefited by the dissemination of these updates respectively ($benefit$). The queuing delay that is experienced by the i^{th} update is $D(u_i) = \sum_{k=1}^{i-1} C(u_k)$ and this queuing delay will result in additional loss of fidelity for all the beneficiaries. Therefore the total delay added due to processing of update u_i at position $i = b(u_i) \times D(u_i)$ and total delay for processing$= \sum_i(\sum_{k=1}^{i-1} C(u_k) * b(u_i))$

We now prove that this sum will be minimum, if $u_1, u_2..u_n$ are such that for each i, $(b(u_i)/C(u_i)) > (b(u_{i+1})/C(u_{i+1}))$. For this consider an ordering wherein, for some i, update u_{i+1} is processed before u_i. In the original schedule delay due to u_i and u_{i+1}

$$D_{orig} = \sum_{k=1}^{i-1}(C(u_k) \times b(u_i)) + \sum_{k=1}^{i}(C(u_k) \times b(u_{i+1}))$$

and in the reverse schedule, the delay added due to u_i and u_{i+1} is

$$D_{inv} = \sum_{k=1}^{i-1}(C(u_k) \times b(u_{i+1})) + \sum_{k=1}^{i-1,i+1}(C(u_k) \times b(u_i))$$

$$D_{orig} - D_{inv} = C(u_i) \times b(u_{i+1}) - C(u_{i+1}) \times b(u_i)$$

which is positive, since $b(u_{i+1})/C(u_{i+1}) > b(u_i)/C(u_i)$, as per our assumption. Therefore any schedule with an order inversion causes a larger total delay, and can be improved upon by removing the inversion, and the schedule with no inversions i.e., one ordered by a score $= (b(u_i)/C(u_i))$, is optimal.

Suppose repository A sends updates to repository B and also to some clients. To parallelize the servicing of dependents by the two servers the updates should be disseminated to B first and then the clients. This implies that pushes to dependents must be scheduled carefully. Suppose there is a set of outstanding dependents $r_1, r_2 \ldots r_n$ to which an update needs to be pushed and let this be the optimal order. Arguing on similar lines as for determining the update processing order , if $b_i(u)$ is the benefit of pushing an update u to the i^{th} dependent r_i, and $c_i(u)$ is the cost of pushing it, the criterion for the optimal pushing order comes out to be the same, i.e., $(b_i(u)/c_i(u))$. Since the costs for each push is the same, the optimal dissemination order is in the decreasing order of benefits $b_i(u)$.

The scheduling schemes mentioned till now assume that given an update u, we can determine the cost of processing the update $C(u)$, and the total benefit of processing the update $b(u)$, and the benefit of disseminating it to a particular dependent i, $b_i(u)$. Since the checking delay is much smaller compared to the computational delay, $C(u)$ depends on the number of computations or pushes that will be required. If Sub_i is the subtree rooted at the dependent i, $b_i(u)$ will be the total number of dependents under Sub_i, who need to be updated about u. These need to be precomputed and stored. Therefore we discretize

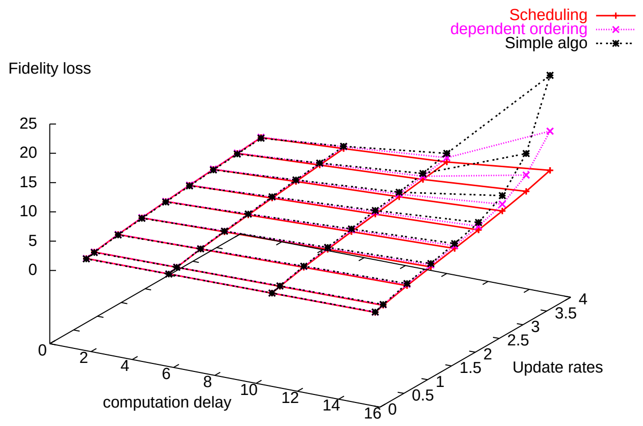


Figure 5: Variation of fidelity loss with update rates (per sec, per data item), computational delay (ms) for T=80%

the update space by mapping it on to the *cr* space. This is automatically done for us by the unique coherence algorithm [26] used for data dissemination, wherein the source has access to all unique coherency requirements and maps the update value u to the maximum coherency requirement violated, say $c_{max}(u)$. Therefore, the mapping can be stored at each repository for each unique coherency requirement value in the subtree rooted at this repository. Therefore
$b_i(u)$ = number of dependents in Sub_i with coherency requirement less that $c_{max}(u)$
and
$b(u) = b(c_{max}(u)) = \sum_i b_i(u)$
$C(u) = C(c_{max}(u)) = total\ number\ of\ immediate\ dependents\ with\ cr < c_{max}(u)$.

Before we discuss the performance improvement due to such informed scheduling, it is important to point out that at higher update rates queue lengths can get very high, resulting in large propagation delays and low fidelity. One way to reduce the effect of such overflow situations is to ignore, i.e., drop, certain updates. As it turns out, our scheduling approach gives us a good criterion to use while dropping updates, namely, based on the importance or the $(benefit/cost)$ ratio of an update. In conjunction with our queuing policy, the processing of updates with a low ratio gets delayed, and it is likely that when a new update to the same data item comes in later, the older one will be dropped.

5.3 Experimental results

It can be seen from Figure 5 that *dependent ordering* has lower loss of fidelity compared to the "simple algorithm" i.e., one without any specific scheduling policy or dependent ordering built into a repository, but *scheduling* performs the same or much better than these two. This difference in performance is substantial, up to 15%, for T=80%, i.e., tight coherency situations. All the results

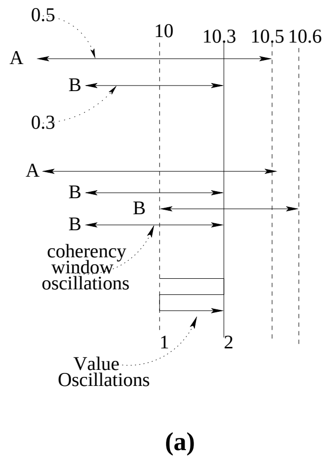


Figure 6: For requests as in Figure 4, Normal method(left): When data value oscillates between 10 and 10.3, the values 10 and 10.3 are successively pushed to B, and its coherency window oscillates.Dependent ordering(right) - Pushing a pseudo value of 10.2 sets the coherency window to (9.9,10.5) and further oscillations between 10 and 10.3 fall within this window and need not be pushed.

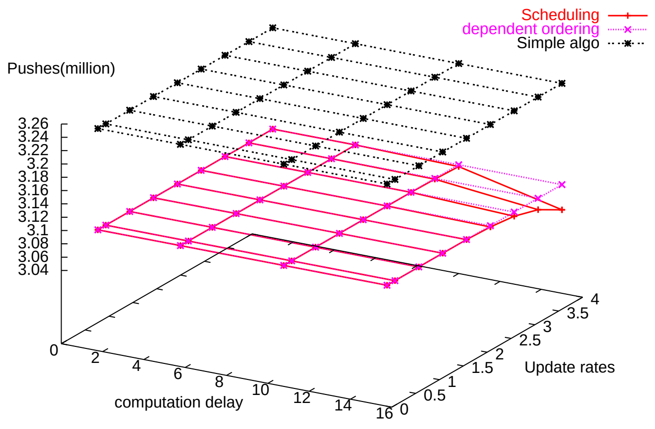


Figure 7: Variation in Number of Messages with update rates (per data item, per sec), computational delay (ms) for T=80%

shown are for 100 data items unless explicitly mentioned.

The performance can be explained by examining the number of pushes. Figure 7 shows the number of pushes for T=80%. As it can be seen, the number of pushes for the dependent ordering case has reduced from 3.245 million pushes to 3.093 million. This is somewhat counter intuitive, because one would expect that since the values pushed are bounded by other dependents as in Eq. (1), a more restricted value is pushed and is likely to require a push earlier than using a simple algorithm. However, if the data item shows an oscillatory behavior as shown in Figure 6(a), sending an restricted value will reduce the number of pushes as shown in Figure 6(b). The use of scheduling improves the performance further because it makes informed decisions regarding the order of updates and the order of pushes. The number of pushes reduces further at high update rates (rates are measured as the average number of updates per data item per sec) and computational delays further to 3.055 million from 3.093 million. This is because the queue starts building up at that point and some of the updates get dropped. That accounts for the fact that the techniques not using scheduling and dropping of updates exhibit a large loss in fidelity at high update rates, while the improved technique has a near linear behaviour even at high loads/update rates.

Graph 8 shows the fidelity loss with better scheduling, across various number of data items. It can be seen that though the fidelity drops with an increase in the number of data items, even at reasonably high update rates and computational delays, the fidelity loss with a large number of data items is within 10%. Thus good scheduling techniques allow the system to scale much better and degrades gracefully.

Finally, we discuss performance results that show how our scheduling technique interacts with the resiliency improvement technique. In Figures 3(b) and (c), the graph marked "scheduling" clearly shows that improvements do occur by as much as 1%, beyond those made possible by the resiliency improvement techniques, especially when the number of data items is not very high.

In summary, the key benefits of maintaining the dependents ordered by crs are: (a) It reduces the number of checks required for processing each update. (b) More often than not, it reduces the number of pushes required as well. This is because this approach disseminates pseudo values to the user, which results in a coherency window covering oscillatory behavior of the data.

Our scheduling approach (a) reduces the overall propagation delay to the end clients, by processing updates which provide a higher benefit at a lower cost earlier, (b) gives a better choice in dropping updates as low score updates may be dropped rather than later arriving ones, and (c) due to a lower propagation delay, a system which uses scheduling scales better and degrades gracefully under unexpected heavy loads.

6 Related Work

Push-based dissemination techniques that have been recently developed include broadcast disks [1], publish/subscribe applications [20, 3], web-based push caching [14], and speculative dissemination [4].

The design of coherency mechanisms for web workloads has also received significant attention recently. Proposed techniques include strong and weak consistency [18] and the leases approach [9, 30]. Our contributions in this area lie in the definition of coherency in combination with the fidelity requirements of users. Coherency maintenance has also been studied for coopera-

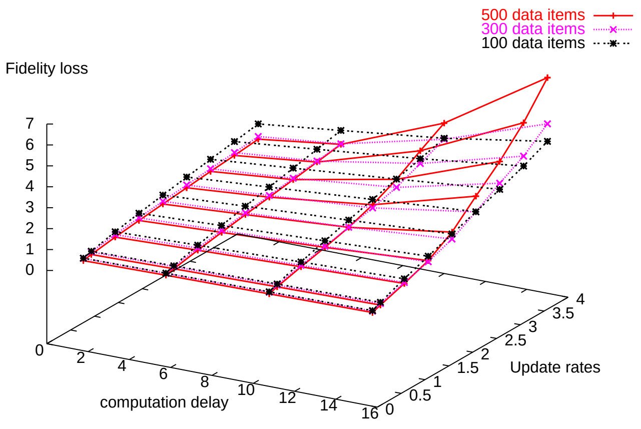


Figure 8: Variation of fidelity with computation delay (ms), update rates (per sec, per data item) for network containing 100 repositories for 100, 300 and 500 data items at T=50%

tive web caching in [29, 28, 30]. The difference between these efforts and our work is that we focus on rapidly-changing dynamic web data while they focus on web data that changes at slower time-scales (e.g., tens of minutes or hours)—an important difference that results in very different solutions.

Efforts that focus on *dynamic* web content include [16] where push-based invalidation and dependence graphs are employed to determine where to push invalidates and when. Scalability can be improved by adjusting the coherency requirements of data items [31]. The difference between these approaches and ours is that repositories don't cooperate with one another to maintain coherency.

Mechanisms for disseminating fast changing documents using multicast-based push has been studied in [25]. The difference though is that recipients receive *all* updates to an object (thereby providing strong consistency), whereas our focus is on disseminating only those updates that are necessary to meet user-specified coherency tolerances. Multicast tree construction algorithms in the context of application-level multicast have been studied in [13]. Whereas these algorithms are generic, the d^3t in our case, which is akin to an application-level multicast tree, is specifically optimized for the problem at hand, namely maintaining coherency of dynamic data.

Several research groups and startup companies have designed adaptive techniques for web workloads [6, 12]. But as far as we know, these efforts have not focused on distributing very fast changing content through their networks, instead, handling highly dynamic data at the server end. Our approaches are motivated by the goal of offloading this work to repositories at the edge of the network.

The concept of approximate data at the users is studied in [23, 22]; the approach focuses on pushing individual data items directly to clients, based on client coherency requirements and does not address the additional mechanisms necessary to make the techniques resilient. We believe that in this sense, the two approaches are complementary since our approaches to cooperative repository based dissemination can be used with their basic source-client based dissemination.

Our work can be seen as providing support for executing continuous queries over dynamically changing data [19, 8]. Continuous queries in the Conquer system [19] are tailored for heterogeneous data, rather than for real time data, and uses a disk-based database as its back end. NiagraCQ [8] focuses on efficient evaluation of queries as opposed to coherent data dissemination to repositories (which in turn can execute the continuous queries resulting in better scalability).

There has also been some work on dissemination in database systems. An architecture for a scalable trigger processing system, and an index structure for it is described in [15]. Given a set of materialized views, [17] focuses on finding the best order to refresh them in the presence of continuous updates, to maximize the quality of data served to users. [2] deals with processing updates in a soft real time system in a manner such that it keeps database ”fresh”, by deciding the order in which updates and transactions are executed. On the one hand, our problem, of determining which update needs to be propagated is simpler because of the numerical nature of the data. This implies that simpler techniques than the ones above, e.g., [15], are sufficient. On the other hand, given the d^3t structure, the scheduling decisions of one repository can have implications for the fidelity experienced way downstream in the tree. Our solutions exploit the simplicity of decision making while catering to the specific characteristics of the d^3t and the semantics of coherency.

Finally, it is important to point out that our work is among the first to directly deal with the problem of failures in disseminating dynamic data by constructing a resilient dissemination network.

7 Conclusions

In this paper, we examined the design of a data dissemination architecture for time-varying data. The architecture ensures data coherency, resiliency, and efficiency. The key contributions of our work are:

- Design of a push-based dissemination architecture for time-varying data. One of the attractions of our approach is that it does not require all updates to a data item to be disseminated to all repositories, since each repository's coherency needs are explicitly taken into account by the filtering component of the dissemination algorithm.

- Design of a mechanism for making the cooperative dissemination network resilient to failures so that even under failures data coherency is not completely lost. In fact, an interesting byproduct of the way resiliency is provided is that even under many non-failure situations, fidelity improves due the resiliency improvement measures.
- The intelligent filtering, selective dissemination, and smart scheduling of pushes by a repository reduces the system-wide network overhead as well as the load on repositories. These in turn improve the fidelity of data stored at repositories. Further advantages accrue when the resiliency enhancement features are combined with the scheduling features.

Whereas our approach uses push-based dissemination, other dissemination mechanisms such as pull, adaptive combinations of push and pull [5], as well as leases [21] could be used to disseminate data through our repository overlay network. The use of such alternative dissemination mechanisms as well as the evaluation of our mechanisms in a real network setting is the subject of future research.

References

[1] S. Acharya, M. J. Franklin, and S. B. Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD Conference*, May 1997.

[2] B. Adelberg, H. Garcia-Molina and B. Kao, Applying Update Streams in a Soft Real-Time Database System *Proceedings of the 1995 ACM SIGMOD, pp. 245 - 256, 1995*

[3] 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. In *International Conference on Distributed Computing System*, 1999.

[4] A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time in distributed information systems. In *International Conference on Data Engineering*, March 1996.

[5] M. Bhide, P. Deolasse, A. Katker, A. Panchgupte, K. Ramamritham, and P. Shenoy. Adaptive push pull: Disseminating dynamic web data. *IEEE Transactions on Computers special issue on Quality of Service*, May 2002.

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

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worell. A hierarchical internet object cache. In *Proceedings of 1996 USENIX Technical Conference*, January 1996.

[8] J. Chen, D. Dewitt, F. Tian, and Y. Wang. Niagracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, May 16-18 2000.

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

[10] A. Fei, G. Pei, R. Liu, and L. Zhang. Measurements on delay and hop-count of the internet. In *IEEE GLOBECOM'98 - Internet Mini-Conference*, 1998.

[11] Z. Fei, A Novel Approach to Managing Consistency in Content Distribution Networks *Proc. of Sixth Int'l Workshop on Web Caching and Content Distribution.*, 2001

[12] 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.*

[13] P. Francis. Yallcast: Extending the internet multicast architecture. http://www.yallcast.com, September 1999.

[14] J. Gwertzman and M. Seltzer. The case for geographical push caching. In *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, May 1995.

[15] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy and J. B. Park and A. Vernon, Scalable Trigger Processing, *In Proceedings International Conference on Data Engineering 1999, pages 266-275.*

[16] A. Iyengar and J. Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[17] A. Labrinidis, N. Roussopoulos, Update Propagation Strategies for Improving the Quality of Data on the Web *In the Proceedings of the 27th International Conference on Very Large Data Bases (VLDB'01), Roma, Italy, September 2001.*

[18] C. Liu and P. Cao. Maintaining strong cache consistency in the world wide web. In *Proceedings of ICDCS*, May 1997.

[19] 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.

[20] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push based distribution substrate for internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[21] A. Ninan, P. Kulkarni, P. Shenoy, K. Ramamritham, and R. Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of WWW10*, 2002.

[22] C. Olston and J.Widom. Best effort cache synchronization with source cooperation. In *Proceedings of the ACM SIGMOD Conference*, June 2002.

[23] C. Olston, B. T. Loo, and J.Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the ACM SIGMOD Conference*, May 2001.

[24] M. S. Raunak, P. J. Shenoy, P. Goyal, and K. Ramamritham. Implications of proxy caching for provisioning networks and servers. In *In Proceedings of ACM SiGMETRICS conference*, pages 66–77, 2000.

[25] P. Rodriguez, K. W. Ross, and E. W. Biersack. Improving the WWW: caching or multicast? *Computer Networks and ISDN Systems*, 1998.

[26] S. Shah, K. Ramamritham and P. Shenoy, Maintaining Coherency of Dynamic Data in Cooperating Repositories, *Proceedings of the 28th Conference on Very Large Data Bases, 2002.*

[27] S. Shah, K. Ramamritham and P. Shenoy, Resilient and Coherency Preserving Dissemination of Dynamic Data Using Cooperating Peers, *Technical Report-1, May 2003, IIT. Bombay.*

[28] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. In *IEEE International Conference on Distributed Computing Systems*, 1999.

[29] J. Yin, L. Alvisi, M. Dahlin, C. Lin, and A. Iyengar. Engineering server driven consistency for large scale dynamic web services. *Proceedings of the WWW10*, 2001.

[30] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchical cache consistency in a WAN. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[31] H.Yu and A.Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of OSDI*, October 2000.