Resilient and Coherence Preserving Dissemination of Dynamic Data Using Cooperating Peers

Shetal Shah * Krithi Ramamritham † Prashant Shenoy ‡

Abstract

The focus of our work is to design and build a dynamic data distribution system that is *coherence-preserving*, i.e., the delivered data must preserve associated coherence requirements (the user specified bound on tolerable imprecision) and *resilient* to failures. To this end, we consider a system in which a set of repositories cooperate with each other and the sources, forming a peer-to-peer network. In this system, necessary changes are *pushed* to the users so that they are automatically informed about changes of interest. We present techniques (i) to determine when to push an update from one repository to another for coherence maintenance, (ii) to construct an efficient dissemination tree for propagating changes from sources to cooperating repositories, and (iii) to make the system resilient to failures. An experimental evaluation using real world traces of dynamically changing data demonstrates that (i) careful dissemination of updates through a network of cooperating repositories can substantially lower the cost of coherence maintenance, (ii) unless designed carefully, even push-based systems experience considerable loss in fidelity due to message delays and processing costs, (iii) the computational and communication cost of achieving resiliency can be made to be low, and (iv) *surprisingly*, adding resiliency can actually improve fidelity even in the absence of failures.

Keywords: Resiliency, Dynamic Data Dissemination, Data Coherence, Cooperation

1 Introduction

The Internet and the web are increasingly used to disseminate fast changing data such as sensor data, traffic and weather information, stock prices, sports scores, and even health monitoring information [18]. These data items are *highly dynamic*, i.e., the data changes continuously and rapidly, *streamed* in real-time, 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 monitoring such data for on-line decision making. The growth of the Internet has made the problem of managing 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 it to geographically closer users. Services like *Akamai* 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). Although such systems scale quite well, when the data is changing rapidly, the quality of service at a repository farther from the data source will deteriorate. In general, replication can reduce the load on the sources, but replication of time-varying data introduces new challenges. Unless updates to the data are carefully disseminated from sources to repositories (to keep them coherent with the sources), the communication and computation overheads involved can result in delays as well as scalability problems, further contributing to loss of data coherence.

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 *coherence requirement (cr)* associated with the data. The coherence requirements associated with a time-varying data item depend on the nature of the item and user tolerances. For example, a user involved in exploiting exchange disparities in different markets or an on-line stock trader may impose stringent coherence requirements (e.g., the stock price reported 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

*Shetal Shah is with Tata Research Development and Design Centre, Pune, currently working at the TCS Lab for Internet Research in Computer Science and Engineering Department, IIT Bombay. Email: shetals@cse.iitb.ac.in

†Krithi Ramamritham is with the Computer Science and Engineering Department, Indian Institute of Technology Bombay, Mumbai - 400 076. Email: krithi@cse.iitb.ac.in

‡Prashant Shenoy is with the Department of Computer Science, the University of Massachusetts, Amherst, MA 01003, USA. Email: shenoy@cs.umass.edu

may be content with a less stringent coherence requirement. The basic framework underlying the coherence model is outlined in Section 2.

The focus of our work is to design and build a dynamic data distribution system that is *coherence-preserving*, i.e., the delivered data must preserve associated coherence requirements, and *resilient*, i.e., the system should be resilient to failures. We consider a system in which the necessary changes are *pushed* to the users i.e., the users are automatically informed about changes of interest, rather than each user independently polling the source(s) for changes of interest. The effectiveness of the system's response to users' requests can then be measured using *fidelity*, which is indicative of degree to which the user's coherence needs are satisfied. Contrary to expectations, we find that push-based systems, unless designed carefully, can experience considerable loss in fidelity due to communication delays and processing costs. Addressing this challenge, even in the presence of failures in the system, is the goal of this paper.

In this paper, we examine techniques to maintain the coherence of time-varying data items at a set of repositories. Each repository is assumed to store a subset of these data items, each of which has a coherence requirement associated with it. A particular focus of our work is to investigate how repositories can cooperate with each other and the source to reduce the overheads of coherence maintenance. To do so, we assume that repositories storing a particular data item are logically connected to form an overlay network that we refer to as a *dynamic data dissemination tree* (abbreviated as d^3t). The source of the data item forms the root of the d^3t. Instead of directly disseminating changes to a data item to all interested repositories, the source pushes these changes only to its children in the d^3t (each child is also referred to as a *dependent* of its parent). Each repository in turn pushes these changes to its dependent repositories. Dissemination using the d^3t incurs two kinds of overheads: i) *Communication delays:* This is the delay incurred in propagating an update from a repository to a dependent and includes all communication related delays, the message processing delays at the source and destination of a message and the delays on all physical links between the two, and ii) *Computational delays:* This is the delay resulting from the computations performed by a repository to send an incoming data change to one of its dependents.

The contribution of this paper lies in providing efficient solutions to the three major issues that need to be solved to effectively deliver dynamic data with high fidelity in distributed systems.

1) Filtering of dynamic data in the overlay network: When disseminating updates to its dependents, a repository needs to be cognizant of their data and coherence needs. We show that (a) it is necessary to place repositories with stringent coherence requirements closer to the source, (b) *cooperation* amongst repositories - where each repository pushes updates of data items to other repositories – helps reduce the overheads for coherence maintenance and (c) a repository may have to receive more than the updates it itself needs so as to meet the coherence needs of its dependents—*even if the coherence needs of its dependents are less stringent than its own.*

2) Construction of an effective dissemination network of repositories: The objective is to construct a d^3t that reduces the overheads while meeting the coherence requirements at all repositories. We develop two different algorithms to meet this objective. Our first approach makes decisions—regarding where to insert a repository in a network— one level at a time, whereas our second approach examines nodes along a chosen path to make this decision. We compare the performance of the two algorithms and *show that it is essential that the d^3t be structured so as to balance the computational and communication costs to achieve high fidelity*.

3) Making the network resilient to failures. Any algorithm for dynamic data dissemination must handle failures of repositories as well as the links connecting the repositories. Our approach 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 coherence that is less stringent than that associated with the parent. This reduces the overheads of providing resiliency, yet allows the dependent to determine if the parent has failed. Upon failure, the back-up parent becomes an alternative parent. Our measures to add resiliency to the dissemination tree result in an interesting and useful side effect: In many of the cases, *adding resiliency improves fidelity even in the absence of failures.*

The rest of the paper is structured as follows. The basic framework underlying the coherence model is outlined in Section 2. We present efficient methods for data filtering based on coherence requirements in Section 3. Section 4 discusses an algorithm for constructing an overlay network for dynamic data dissemination followed by the experimental evaluation in Section 5. Section 6 presents another algorithm to build the network and compares it that presented in Section 4. Techniques to add resiliency to the network and their performance evaluation are given in Section 7. Section 8 presents related work, and Section 9 presents our conclusions and directions for future work.

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

Figure 1: Basic Framework

2 Problem Formulation and Background

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 coherence requirements. Coherence requirement (c^u) associated with a data item denotes the maximum permissible deviation of the user (u)'s view from the value of data d at the source. Generally, c^u 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., a stock price should never be out-of-sync by more than ten cents). 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 coherence 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 coherence, we should have $|U_x(t) - S_x(t)| \leq c^u$.

The issue of what user should connect to which repository 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., $|R_x(t) - S_x(t)| \leq c^u$. 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 observation. The goal of a good coherence mechanism is to provide high fidelity at low cost. Although Figure 1(b) shows a single data repository, the coherence 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, 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 a tree rooted at the source and consisting of repositories $R_1, \ldots, R_n$. We refer to this tree as the *dynamic data dissemination tree*, or d^3t, for x. 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. We assume that the architecture uses the *push* approach for disseminating updates—the source pushes updates to its dependents in the d^3t, which in turn push these changes to their dependents and the end-users. Not every update needs to be pushed to a dependent—*only those updates necessary to maintain the coherence requirements at a dependent need to be pushed*. To understand when an update should be pushed, let c^p and c^q denote the coherence requirements of data item x at repositories P and Q, respectively. Suppose P serves Q. To effectively disseminate updates to its dependents, the coherence requirement at a repository should be *at least as stringent as those of its dependents*:

$$c^p \quad \leq \quad c^q \tag{1}$$

Given the coherence requirement of each repository and assuming that the above condition holds for all nodes and their dependents in the d^3t, we now derive the condition that must be satisfied during the dissemination of updates. Let $x^s_i, x^s_{i+1}, x^s_{i+2}, \ldots x^s_{i+n} \ldots$ denote the sequence of updates to a data item x at the source S. This is the data stream x. Let $x^p_j, x^p_{j+1}, \ldots$ denote the sequence of updates received by a dependent repository P. Let x^p_j correspond to update x^s_i at the source and let x^p_{j+1} correspond to update x^s_{i+k} where $k \geq 1$. Then, $\forall m, 1 \leq m \leq k-1, |x^s_{i+m} - x^s_i| < c^p$. Thus, as long as the magnitude of the difference between last disseminated value and the current value is less than the coherence requirement, the current update is not disseminated (only updates that exceed the coherence tolerance c^p are disseminated). In other words, the repository P sees only a "projection" of the sequence of updates seen at the source. Generalizing, given a d^3t, each downstream repository sees only a projection of the update sequence seen by its predecessor.

Dissemination techniques should construct these projections efficiently because, even if all the necessary updates are propagated by a repository to its dependents due to the non-zero computational and communication delays in real-world networks and systems, data at a dependent will experience loss of coherence. Thus, it is impossible to achieve 100% fidelity in practice, even in expensive dedicated networks. The goal of our cooperative repository architecture is to achieve high fidelity in real-world settings where computational and communication overheads are non-negligible.

3 Filtering Data as it is Disseminated

Assuming that a d^3t has been constructed for data item x, consider a source S that disseminates x to a repository P, which in turn disseminates x to a dependent repository Q. In this section, we answer the following question: When should a a repository/source forward an update to its dependent?

Recall from Eq. (1) that to effectively disseminate updates, we require that the coherence requirement at P should be at least as stringent as that of Q.

Let x_i^s, x_{i+1}^s, x_{i+2}^s ... denote a sequence of updates to x at the source S. Let x_j^p, x_{j+1}^p, x_{j+2}^p ... denote the updates received by P and x_k^q, x_{k+1}^q, x_{k+2}^q .... denote the updates received by Q. Since $c^p \leq c^q$, the set of updates received by Q is a subset of that received at P, which in turn is a subset of unique data values at the source. Specifically, an update x_j^p received by P is forwarded to Q if

$$\mid x_j^p - x_k^q \mid \quad \geq \quad c^q \tag{2}$$

where x_k^q denotes the previous update received by Q. Intuitively, Eq. (2) indicates that any update that violates the coherence requirements of Q is forwarded to Q. We now show that this is a necessary but not sufficient condition for maintaining coherence at Q. Suppose x_i^s, x_j^p and x_k^q represent the value of x at S, P and Q, respectively. Let the next update at S be x_{i+1}^s such that

$$\mid x_{i+1}^s - x_j^p \mid \quad < \quad c^p \tag{3}$$

$$\mid x_{i+1}^s - x_k^q \mid \quad \geq \quad c^q \tag{4}$$

Thus, the next update is of interest to repository Q but not to P. Since S is logically connected only to P, if S does not disseminate this update to P, then Q, will miss this update (causing a violation of Q's coherence requirement). Figure 2 provides an example of this situation. Thus, even under ideal conditions of zero processing and communication delays, a dissemination technique that uses solely Eq. (2) to disseminate updates might not provide 100% fidelity (indicating Eq. (2) is not a sufficient condition to maintain coherence). Hence, dissemination algorithms need to be developed carefully to avoid such *missed updates* problem.

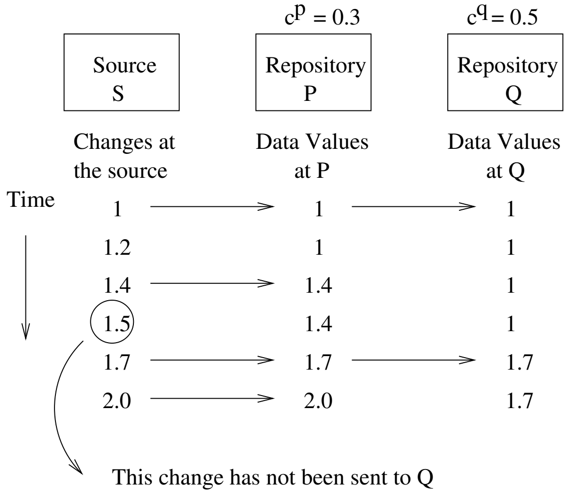


Figure 2: Need for Careful Dissemination of Changes

Next, we present two approaches to address this issue and also examine the entailed overheads.

3.1 Repository-Based (Distributed) Approach

The missed updates problem described earlier occurs when an update x^s_{i+1}, where $x^s_i < x^s_{i+1} < x^s_i + c^p$, satisfies both Eqs. (3) and (4). From these equations, we get,

$$| x^s_{i+1} - x^p_j | - | x^s_{i+1} - x^q_k | < c^p - c^q \quad (5)$$

Also,

$$| x^p_j - x^q_k | < c^q \quad (6)$$

Let us consider the following cases for Eq. (5)

Case 1: Let, $x^s_{i+1} > x^p_j$ and $x^s_{i+1} > x^q_k$. In this case it is trivial to see that Eq. (5) reduces to:

$$c^q - | x^p_j - x^q_k | < c^p \quad (7)$$

Similarly for the case of $x^s_{i+1} < x^p_j$ and $x^s_{i+1} < x^q_k$.

Case 2: Let, $x^s_{i+1} > x^p_j$ and $x^s_{i+1} < x^q_k$. The above condition can be combined into

$$x^q_k > x^s_{i+1} > x^p_j \quad (8)$$

Now from Eq. (8), Eq. (3) and Eq. (4) we get $x^q_k - x^p_j > c^q$ which is in contradiction with Eq. (6). Hence this case is not possible at all. We can similarly argue for the last case where $x^s_{i+1} < x^p_j$ and $x^s_{i+1} > x^q_k$. Hence, we can see that Eq. (5) reduces to Eq. (7).

Eq. (7) represents the additional condition that must be checked by any repository P to see if an update should be disseminated to its dependent Q. Note that this applies even to the source, i.e., when P is the source. Thus, the dissemination technique propagates an update x^p_j received by P to dependent Q if either Eq. (2) or (7) is satisfied. In the example illustrated in Figure 2, such a technique would propagate the update corresponding to value 1.4 from P to Q (since it satisfies Eq. (7)). Consequently, the subsequent increase in value to 1.5 does not result in a violation at Q. Note that the update of 1.4 is not strictly required as per the coherence requirement of Q (Eq. (2)), but is essential to prevent the missed updates problem.

It is easy to show that if a repository makes dissemination decisions based on Eqs. (2) and (7), then 100% fidelity will result in the absence of delays: In addition to P and Q, consider repository R such that P, Q, and R form a chain. If Eqs. (2) and (7) hold between P and Q as well as between Q and R, they will also apply between P and R. Generalizing, it will apply between source S and any repository R. Given that S has complete coherence, $c^s = 0$ and the claim that the two equations guarantee 100% fidelity for all repositories follows.

3.2 Source-Based (Centralized) Approach

In this approach, the source maintains a list of all the unique coherence requirements for a data item x specified by various repositories. For each such coherence requirement, the source also tracks the last update disseminated for that coherence requirement. Upon a new update, the source examines for each unique coherence requirement c the last update sent for that c. It determines all c's that are violated by the update. The update is tagged by the maximum such coherence requirement c_max and the tagged update is then disseminated through the d^3t. The source also records this data value as the last update sent for all c's that are less than or equal to c_max.

Each repository receiving the update forwards it to all dependents that (i) are interested in the data item, and (ii) have a coherence requirement less than or equal to the tagged value. As sketched below this filtering algorithm achieves a fidelity of 100% (in the absence of delays).

Consider a data item x with a value of v. Let the coherence requirement at repository P be c. Suppose that an update causes the value of x to change to v'. If $|v - v'| < c$ then clearly the algorithm works (since no action is necessary for P). Let $|v - v'| \geq c$. Then the coherence requirement has been violated for c. Hence, an update with the new value v' is tagged with $c' \geq c$ and disseminated through the d^3t. Consider the path from the source to repository P. As per Eq. (1), the coherence requirement for every repository on this path is at least c. Consequently, every repository on this path receives the update and disseminates it to its dependent, until it reaches P. Thus, P receives every update that exceeds its coherence requirement, resulting in a fidelity of 100%. Since this argument holds for any repository P, the approach can achieve perfect fidelity at all repositories in the absence of communication and computational delays.

We now discuss the overheads of this approach. Since this approach disseminates updates only when necessary it makes efficient use of the communication resources. In this algorithm, the source finds the maximum coherence value, if any, affected by an update. A large number of unique c values can result in large computational and space overheads at the source. This may affect the scalability of the source as compared to the distributed repository based filtering approach. We study this issue in Section 5 where we evaluate the overheads with real world dynamic data.

4 What Should the Logical Structure of the d^3g Be?

In this section we describe an algorithm to build the overlay network. Initially, the overlay network consists of a set of sources $S_1, \ldots, S_n$. The repositories are then *inserted* into the network, one by one. For simplicity, we describe the algorithm assuming the following scenario: When a repository Q wishes to enter the network, it specifies, the list of data items of interest, their c values, and it's degree of cooperation. By degree of cooperation we mean, the maximum number of dependents the repository is willing to serve. The sources satisfy the needs of Q. If a repository's data needs change or its data coherence needs change, then to handle the changed requirements, the algorithm is reapplied. We do not go into the details of this step in this paper.

Our algorithm constructs a single dynamic data dissemination graph, d^3g, during a single traversal of the repository network starting with the sources $S_1, \ldots, S_n$. For any particular data item x, the d^3g reduces to a tree (i.e., the d^3t) that consists of the paths along which an update to x is disseminated. The d^3g is the union of the d^3ts of the data items of interest.

We call our algorithm LeLA (Level by Level Algorithm), because it looks for a position for Q in the current d^3g, level by level. The sources are at level 0, the repositories to which the sources disseminate data are in level 1, the dependents of repositories at level l are at level $l+1$, and so on.

Starting at level 0, repositories at the current level are examined for their suitability to serve the new repository Q, that is, whether Q can become the dependent of a set of repositories at that level. This decision is made by a specially designated *load controller node* at each level. One of the sources S_i vacuously serves as the load controller for level 0. This load controller examines if Q can be served by the sources, if not, the request is passed to load-controller of the next level.

The function of the load controller at a level l is to find a set of suitable $parent$ repositories, at that level to serve the data needs of Q. For each repository in its level, the load controller calculates a *preference factor*. The smaller this factor, the more preferred the repository is to be a parent of Q. We will explain the calculation of the preference factor shortly. For now, let us assume that this factor has been calculated for each repository at level l. We consider all repositories with preference factor within 5% of the smallest preference factor for the current level as potential parents. This allows multiple repositories to become parents of Q, each serving a different subset of data needed by Q. A potential parent P can serve a data item x to Q, if both Q and P are interested in x and if the coherence requirement of P for x is at least as stringent as that of Q. If more than one repository can serve a data item x to Q then the more preferred among these is asked to serve x to Q.

It is quite likely that Q might want some data items, say, $x_i, ..., x_k$, which are not served by any of the potential parents. The most preferred repository P is made to serve $x_i, ..., x_k$ to Q. This process of *augmenting* a parent's data requirements—to serve the needs of a new child—can have a cascading effect: For each of these data items, P checks if any of its parents are serving it and if so, requests the parent for service, else it randomly selects one of its parents and asks it to service the data item to P at the coherence required by Q. This is continued all the way up the d^3g till there is a path from the source to Q for those data items.

The following factors are used to determine the preference factor of a node:

1. *Data Availability Factor:* The number of data items that a parent can serve Q, with its current data and coherence requirement.
2. *Computational delay Factor:* The larger the computational delay incurred at a parent P to disseminate a data change to its dependents, the less preferred it is. We approximate this delay by the number of dependents P has: On an average, more the dependents P has, greater will be the computational delays encountered by Q to get a data update from P.
3. *Communication delay Factor:* Parents which have a large communication delay with Q are less preferred.

Since we want to choose parents such that the delays are low and the data availability is high, we calculate the preference factor as: $\frac{communication\ delay(P,Q) \times number\ Dependents(P)}{number\ of\ data\ items\ P\ can\ serve\ Q}$.

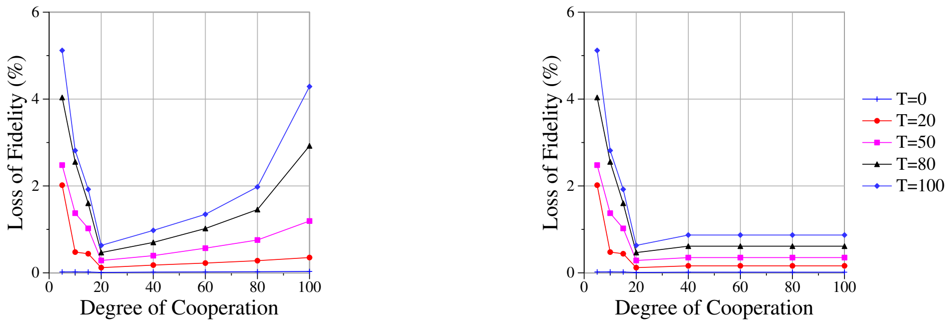
A repository is considered as a candidate for becoming a parent only if the number of dependents currently served by it is smaller than its degree of cooperation. The degree of cooperation offered by a repository P is its maximum fan out in the d^3g. As long as there are repositories with less dependents than the degree of cooperation specified, the load controller will find suitable parents from its level. To this end, a load controller's view of a repository at its level is updated whenever a new repository becomes its dependent. If all of the repositories have as many dependents as the degree of cooperation, the load controller passes the request to the load controller of the next level.

4.1 How Much Should a Repository Cooperate?

We saw that each repository that requests insertion into the cooperative repository network, indicates how many dependents it is willing to support, namely, its degree of cooperation. In this section, we argue the need to judiciously choose the degree of cooperation and present a heuristic to do so.

A high degree of offered cooperation implies that a repository is willing to take on increased responsibilities, which can help reduce source overload and potentially improve fidelity. But, a repository offering a high degree of cooperation can also indirectly lead to a loss in fidelity - it may just transfer the source load onto itself. A greater degree of cooperation increases the computational delay but reduces the end-to-end network delay (by virtue of reducing the path length from the source to the farthest repository). On the other hand, a small degree of cooperation reduces the computational delay at a repository but increases the end-to-end network delays. In one extreme case, if the degree of cooperation is reduced to one, the d^3t becomes a linear chain of repositories with a large network delay. The other extreme results in the source serving all the repositories resulting in large computational delays. To maximize fidelity, the d^3t should be constructed such that the *sum of two delay components is minimized*.

As shown in Figure 3(a), for a given set of repositories, the variation in the (loss of) fidelity with increasing degree of offered cooperation exhibits a U-shaped curve. The left end of the x-axis corresponds to the d^3t being a chain and the right end to the case where a source directly disseminates updates to all its dependents. This curve portrays the results for different values of a parameter T—which encodes the stringency of the overall coherence requirements of repositories. (Section 5 presents details of how these curves were derived.) For now, it suffices to know that $T = 100\%$ signifies that all repositories have very stringent coherence requirements. As we can see from the plots, except when repositories do not have stringent coherence requirements, the choice of the degree of cooperation offered does make a difference on the achieved fidelity.



(a) With Uncontrolled Cooperation

(b) With Controlled Cooperation

Figure 3: Performance of LeLA

The point where the loss in fidelity is minimized depends on the minimum total network and computational delays incurred by the d^3t. In the falling part of the U-shaped curve, the communication delays dominate and in the rising part, the computational delays dominate. The figure shows that *arbitrarily increasing the degree of cooperation can, in fact, be detrimental to fidelity.* Hence, in a system where communication delays dominate, it is prudent to use a high degree of cooperation. On the other hand, if computational delays dominate, then a small degree of cooperation should be chosen. In other words, the degree of cooperation should be directly proportional to the communication delays and inversely proportional to the computational delays. This results in the following heuristic to compute the degree of cooperation:

$$\min\left(\frac{1}{C} \times \frac{average\ comm\_delay}{average\ comp\_delay}, offered\ degree\ of\ cooperation\right) \tag{9}$$

where $average\ comm\_delay$ and $average\ comp\_delay$ denote the average communication delay from one repository to another and the average computational delay in disseminating an update from one repository to its dependent, respectively. The above formula assumes that on average, only $C\%$ of the dependents of a node would be interested in an update. Thus, the above formula allows us to set the degree of cooperation depending on the expected overheads. In Section 5.2, we study the effectiveness of this formula.

5 Experiments and Results

In this section, we demonstrate the efficacy of our techniques through an experimental evaluation. In what follows, we first present the experimental methodology and then the experimental results.

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. (*Min* and *Max* refer to the minimum and maximum 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 rarely change faster 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

Repositories – Data, Coherence and Cooperation characteristics: We simulated the situation where all repositories accessed data kept at one or more sources. Each repository requests a subset of data items, with a particular data item chosen with 50% probability. A coherence requirement c is associated with each of the chosen data items. We use different mixes of data coherence. 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). At each repository, $T\%$ of the data items have stringent coherence requirements (the remaining $(100 - T)\%$, of data items have less stringent coherence requirements). We also present results where the data item chosen for a repository follows a Zipf distribution. This simulates a situation where some of the data items served by each server are needed by almost all the repositories, i.e., they are highly popular.

Physical Network – topology and delays: The physical network consists of nodes (routers and repositories) and links. The underlying router topology was generated using BRITE (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 them to the closest router. For each repository, the set of data items of interest is first generated and then coherencies are chosen from the desired range depending on the value of T.

For our performance measurements, 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 coherence mix, was varied from 0 to 100. Each data item is served by only one source.

Node-node communication delays are derived from a heavy tailed Pareto [22] 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. $\bar{x}$ was set to 1.5 ms (milli secs) and x_1 was 0.2 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 [8]. We also present the results obtained at higher link delays.

A packet (containing a data update) which comes to a repository is first queued. The repository then checks which of its dependents are interested in the update and for the interested dependents, it pushes this update to them. Queuing delay is the delay encountered by the packet while it is waiting in the queue to be processed by the node. Checking delay is the delay experienced when the repository is checking if a change is of interest to a dependent. Depending on the nature of the queries, this delay can be short for a simple check to the tune of a few tens of milliseconds for some complex query processing [6, 16]. We derive the checking delay for each data item for each repository from a heavy tailed Pareto distribution where $\bar{x}$ was 5 ms and x_1 was 1 ms. The average checking delay was around 4 ms. The pushing delay was also derived using a Pareto distribution where $\bar{x}$ was 1 ms and x_1 was 0.125ms. We also experimented with other check and push delays. To model enterprise class source servers, computational overheads for the sources were set to 25% of those of the repositories.

Simulation Procedure: After generating the physical network topology, we generate the topology of the d^3t using the technique discussed in Section 4. The simulation of data dissemination is then done, using the algorithms discussed in Section 3.

Metrics: The key metric for our experiments is the *loss in fidelity* of data. Recall from Section 2 that fidelity is the degree to which a user's coherence requirements are met and is measured as the total length of time for which the inequality $|R(t) - S(t)| \leq c$ holds (normalized by the total length of the observation). 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 $(100 - \text{fidelity})\%$. All the experiments were run multiple times to obtain a 90% confidence interval whose width was 10% of the mean.

5.1 Experimental Results

5.1.1 Baseline Results - With and Without Cooperation

Our first experiment examines the efficacy of the d^3t construction algorithm LeLA. We used the source-based algorithm as the baseline data filtering algorithm. For seven different T values, we vary the degree of cooperation offered from 1 to 100 and measure the efficacy of the resulting d^3t in providing good fidelity. Note that in the presence of the non-zero communication delays, the structure of the d^3t has a significant impact on fidelity. The larger the end-to-end delay, the greater the loss in fidelity. Figure 3(a) shows that there is a significant loss of fidelity at low values for the degree of cooperation. As the number of allowable dependents of a repository (i.e., the maximum degree of cooperation offered) is increased, the loss in fidelity decreases to a minimum and then starts increasing again. This is because when the number of permitted dependents is large, the source serves most repositories directly and the d^3t effectively reduces to a one-level tree with most repositories becoming a direct dependent of the source. Note also that in Figure 3(a), as the fraction of data items with stringent coherence tolerances decreases, the gradient of the loss in fidelity also decreases. Our experiments showed that if the source is entrusted with the task of disseminating updates directly to repositories, then there is a loss in fidelity, regardless of other system parameters. When the number of data items to be handled is large, the computational delays at the source will adversely affect the scalability of the source. This effect will be pronounced when all repositories desire their data at high coherence (as indicated by the T=100% graph).

5.1.2 Improvement in Fidelity when Coherence Requirements are used to Filter Updates

In this section, to demonstrate that to achieve high fidelity only updates of interest should be disseminated by a repository to its dependent we compare our approach to a system where *all* updates to a data item are disseminated to repositories interested in that data item. Such a system is emulated by simply using a very stringent coherence tolerance (T=100%) causing all updates to be disseminated. We compare this system to one where (some of) the coherence requirements are not stringent (T<100%). Less stringent c's result in filtering and selective forwarding of updates.

As can be seen from Figure 3(a), the approach that disseminate all updates (corresponding to T=100%), results in worse fidelity across the complete range of offered values of the degree of cooperation. This is because the latter approach disseminates more messages, which increases the network overheads as well as computational delays at repositories, resulting in greater loss in fidelity. In contrast, intelligent filtering and selective dissemination of updates based on data's coherence requirements can reduce overheads and improve fidelity.

We also compared the performance of the centralized and distributed filtering algorithms presented in Section 3. Our results (not shown here) show that (i) with the centralized filtering approach, the source does nearly 50% more checks of incoming data updates to determine if the data updates needs to be disseminated to its dependents, and (ii) both approaches send around the same number of messages through the system and as discussed in Section 3, both approaches guarantee 100% fidelity. So the distributed approach is preferable.

These results clearly show that, as long as there is some data with stringent coherence requirements, it is important for repositories to cooperate with one another to improve fidelity. Moreover, it is inappropriate to use a very large number of resources towards cooperation. We study the effect of setting the "optimal" level of cooperation in the next section.

5.2 Effect of Controlled Cooperation on Fidelity

We repeat the scenario whose results were depicted in Figure 3(a), but with the degree of cooperation chosen as per Eq. (9). That is, irrespective of how many cooperative resources a node has, Eq.(9) limits the number of resources exploited. As shown in Figure 3(b), the behavior becomes an L shaped curve, that is, after the limit, loss of fidelity stabilizes.

With controlled cooperation in effect, we studied the impact of communication and computational delays on fidelity. The results [24] show that we can counter the effect of large delays in the system by adjusting the degree of cooperation as per Eq. (9). We studied the sensitivity of our results to the constant C. We ran the experiments on different traces for 100 data items to determine the sensitivity of C to a chosen trace. In general, the resulting fidelity is insensitive to the value of C if C is in the range $20 - 50$. Variation in fidelity loss in such cases is only around 1%. In general, the results also show that, using our approach, high fidelity can be obtained even if a repository incurs large computation costs (example, if we extend our approach to execute general continuous queries [6]) or when data sizes are large, in which case the communication delays will be larger.

This clearly demonstrates the benefits of choosing the degree of cooperation based on system overheads for providing high fidelity. We performed some sensitivity experiments which are presented in [24]. Our experiments indicate that the choice of degree of cooperation has a larger impact on performance than the specific formula used to calculate the preference factor.

5.3 Scalability of LeLA

We studied the effect of increasing the number of repositories on fidelity in [24]. We observed that even when the number of repositories grew from 100 (for the base case) to 300 (and with that the total number of nodes in the system grows from 700 to 2100 nodes), the increase in the loss in fidelity with controlled cooperation was observed to be less than 5%.

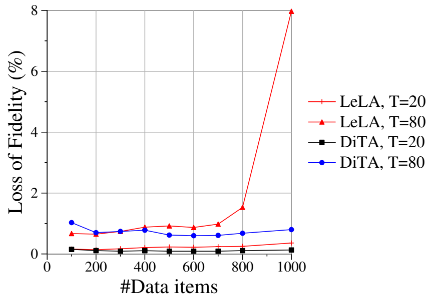


Figure 4: Performance of DiTA versus LeLA for Different Number of Data Items

However, when we increased the number of data items served by the servers was increased and hence the number of data items required by a repository, the loss in fidelity increased considerably, even with controlled cooperation. Figure 4 shows the performance of LeLA with controlled cooperation with increasing number of data items. As can be seen in the graph, for T=80% and 1000 data items, the loss in fidelity is almost 8%. (The curves marked $DiTA$ will be explained in the next section.) In LeLA, if a repository with loose coherence requirements becomes a parent of a repository with stringent coherence requirements, it will have to process many more packets than it needs and this in turn will also increase the delays experienced in the system. To overcome this, repositories with stringent coherence requirements should be kept closer (at a smaller depth) to the source. Next we describe an algorithm that does this.

6 DiTA: *D*ata *i*tem at a *T*ime *A*lgorithm

In $DiTA$, repositories with more stringent coherence requirements are placed closer to the source in the network. As mentioned earlier, the motivation is that such repositories will in most cases get more updates than those with looser coherence requirements. By placing them closer to the source, we can reduce the number of messages in the system and improve fidelity. Since a repository will typically need multiple data items, each with a different coherence requirement, we build a d^3t for each data item. The position of a repository in a d^3t will be governed by its coherence needs for that data item. 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.

As with LeLA, we do not want to overload a repository; consequently we place a limit on the number of unique $<$child, data item$>$ pairs that it can serve. We assume in DiTA that if a repository requests n data items, it has the resources to have at least n unique pairs associated with 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 of x for insertion. When the source gets the request, it checks if it has enough resources to serve x to R. If it has the resources or if the d^3t consists of only one node, i.e., the source, R is made a child of the source in the d^3t for x. If the source does not have the resources, as described next, it determines the most suitable subtree rooted at its children 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 children 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 until 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. Coherence 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.

We refer to the above algorithm as Data-$i$tem-at-a-Time-Algorithm ($DiTA$). DiTA requires very little book-keeping and, experimental results, discussed next, show that it indeed produces d^3ts that deliver data with high fidelity, and is almost an order of magnitude better than LeLA.

6.1 Comparison: DiTA versus LeLA

We can clearly see from Figure 4 that DiTA does much better than LeLA. Specifically, for T=80%, whereas DiTA has around 1% loss of fidelity, LeLA has between 1 and 8% loss. For T=20%, the loss for DiTA is lower than that of LeLA. In DiTA a parent serves one data item to a child on an average whereas in LeLA a parent can serve many more items to a single child. If we take the number of unique <child, data item> pairs to be the fanout of a node, then the average fan-out in LeLA is higher. This amounts to less work done at a node in DiTA and this is a primary contributor to DiTA's superior performance.

Since each node in DiTA does less work than its counterpart in LeLA, the side-effect of this is that the height of the dissemination tree in DiTA is expected to be more than that in LeLA. Thus, only when link delays dominate, can LeLA be expected to have an edge. This can be seen for a smaller number of data items, e.g., 100, where DiTA shows a higher loss in fidelity. In DiTA, the bound placed on the number of data items that are served by a repository increases with the increase in the number of data items needed by the repository. For a smaller number of data items, e.g., 100, the computational delays experienced by the system are small leading to dominance of link delays and hence an increased loss in fidelity. We also experimented with varying link and computational delays in [25]. We found that the behavior of LeLA was better (by just 1.5%) than that of DiTA for very high link delays (average 110ms) and for negligible computational delays (0.5 ms). For all other delays, DiTA does substantially better than LeLA (difference in loss in fidelity is 1-3%). Fortunately, the high link delays we experimented with are not very common on the internet [8], and hence, DiTA is preferable in practice.

7 Enhancing the Resiliency of the Repository Network

Since dynamic data is increasingly used in on-line decision making, there is a need to have fault tolerant dynamic dissemination systems. 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, we need to carefully balance these tradeoffs: 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 back-up parents, so that the resulting overheads do not lead to loss of fidelity during normal operations—the back-up parent serves data with a coherence $> c$.

Let P serve data item d to Q. Q wants the data item at coherence c. Let B be the back-up parent serving Q at c^b. Once we fix the coherence c^b at which B serves a dependent Q we can calculate the expected number of updates lost 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 P fails. A possible embellishment to address this is to make P send periodic "I am alive" messages.)

Once P fails, Q requests B to serve it the data at c. Once 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 coherence, to a dependent even when a parent fails. We now elaborate upon the choice of (a) the coherence associated with the back-up parent, and (b) the back-up parent itself.

7.1 Choosing Coherence of Back-up Parent Using a Probabilistic Model

For the sake of simplicity, we set c^b, the coherence maintained by the back-up 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 on average in case of failure of P. If k is small and in particular if $k = 1$, then both the parent and the back-up parent will send all updates of interest to the child 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. Hence, it is necessary to find out what value to set k as. This depends on the acceptable overheads imposed on the back-up parent and the acceptable number of missed messages. To calculate the number of missed messages, we have two options: (1) observe sample runs, or (2) develop an analytical model. We present a technique to analytically calculate the expected number of updates missed as a function of k. To simplify the treatment, we assume that the data values change in either direction (i.e., increase or decrease) with uniform probability. No assumptions are made about the unit of change or the time taken for a change.

Let $k = 2$. Now suppose P fails. This failure is detected when Q gets 2 successive messages from B without getting a message from P in between. Now we calculate the expected number of updates missed from the time of P's failure till an update is received from B plus the expected number of updates missed till the next update from B. Since the second term is as big as the first, we focus on the second term and upper bound the first with the second. Let the current value be v_0. The expected number of updates missed before the value changes by $2c$ is the expected number of changes of value c before a change of value $2c$. This can be modeled as the Markov Chain given in Figure 5. Starting at v_0, we

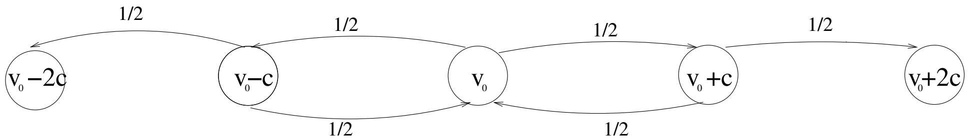
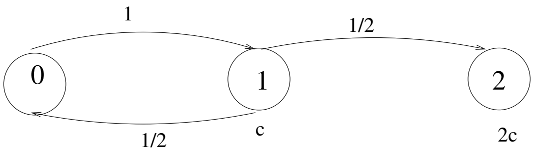
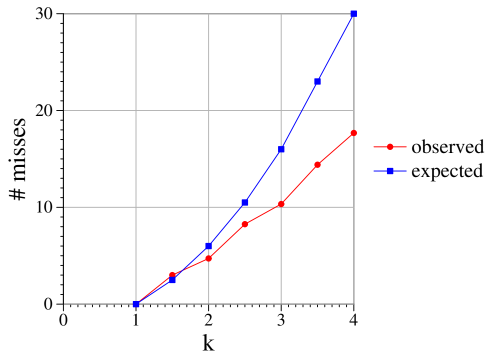


Figure 5: Markov Chain – Changes of 2c

need to know the expected number of steps taken before we reach state $v_0 + 2c$ or $v_0 - 2c$. Since the states $v_0 + c$ and $v_0 - c$ are similar we can merge them together, so also the states $v_0 + 2c$ and $v_0 - 2c$ to get the Markov Chain as shown in Figure 6(a).



(a) Markov Chain with Merged States



(b) Expected versus Observed # Misses

Figure 6: Effect of Choice of k

The expected number of steps taken to go from 0 to $2c$ is given by $X_{0,2}$. Therefore, the expected number of updates from the time of failure till R gets 2 successive updates from B is $2X_{0,2}$ where $X_{0,2} = X_{0,1} + X_{1,2} = 1 + X_{1,2}$. From state 1, with probability $\frac{1}{2}$ we reach state 0 and with probability $\frac{1}{2}$ we reach state 2. Hence, $X_{1,2} = (1/2) * 1 + 1/2 * (X_{0,2} + 1)$; $2X_{1,2} = 2 + X_{0,2}$; $X_{1,2} = 3$; $X_{0,2} = 4$.

For every 4 updates that P sends, B will send 1 update, on an average. $\therefore \#Misses = 8 - 2 = 6$. Similarly, for any k: $X_{0,k} = 1 + X_{1,k}$. $X_{i,k} = \frac{1}{2}(1 + X_{i-1,k}) + \frac{1}{2}(1 + X_{i+1,k})$; $\quad 1 \leq i \leq k - 1$. $\therefore X_{i,k} = 1 + \frac{1}{2}X_{i-1,k} + \frac{1}{2}X_{i+1,k}$. Solving this we get: $X_{i,k} = k^2 - i^2$. And hence, $X_{0,k} = k^2$.

We also calculated the number of actual misses for different values of k experimentally. Essentially, we calculated the number of updates that a repository got from the real parent between two consecutive updates from the back-up parent. The number shown in Figure 6(b) is averaged over 100 repositories and 100 different traces and is plotted with $(2k^2 - 2)$ against k. For traces that do not exhibit uniform change, the expected number of misses may vary considerably from that calculated above. Further, even $(2k^2 - 2)$ is likely to be pessimistic as can be seen in the figure. We can choose a value for k, depending on the number of updates a child may be willing to miss. Given the small number of missed updates for $k = 2$, we chose this value of k in our experimentation.

7.2 Determination of Back-up Parents

For each of its data items, repository Q requests for back-up parents as follows: Let P be the parent of Q in the d^3t for data item x. Let c^b be $k * c$.

7.2.1 LeLA

The load controller at P's level searches for a back-up parent for Q for x at its level. Any repository at the same level that can serve Q at coherence $k * c$ is made the back-up parent for Q. However, if no such repository is found, the load controller of the previous level tries to search amongst the repositories at its level for a suitable back-up parent: we keep going up level by level in search of an eligible back-up parent. If none is found, the source is made the back-up parent for Q for x. We would like to mention here that we do not find a back-up load controller.

7.2.2 DiTA

Consider the siblings of P. If P does not have any siblings, then consider the siblings of the 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 coherence at which Q wants x from B is less than the coherence at B, the parent of B is asked to serve x with the required tighter coherence to B. Note that the coherence 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 coherence requirement of Q.

An advantage of choosing a sibling, as opposed to any other repository in the tree, is that the change in coherence requirement is not percolated higher up. However, choosing a sibling might not be advantageous all the time. If the ancestor of P and B is heavily loaded then the delay due to loading 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 chain of repositories, then the source might finally become the back-up parent of Q for x.

In both the algorithms, back-up dependents are processed after the real dependents at any repository. If a dependent gets the same update from both its parents, then the update which reaches the dependent later is discarded.

7.3 Modeling of Repository Failures and Recovery Times

We used a heavy-tailed Pareto distribution to model the time between failures since it seems to follow the trends reported in [12]. (Since absolute numbers were not specified, we could only model based on the trend observed.) A few repositories experienced a relatively large number of failures, whereas a large number experienced few failures while a few did not experience any failure in the time duration of our runs (which lasted 3 hours). Specifically, about 50-60% of the failed repositories experienced at least one failure. About 1% repositories experienced at least 4 failures while 20% experienced 2 or more failures and the rest failed once.

Our recovery model is also based on the observations of [12]. The distribution for recovery times (in secs) is given by $2 * (900)^r$, where r is a random number between 0 and 1. The minimum recovery time was 2 sec and the maximum was 1800 sec (30 minutes). Approximately 10% of the failures lasted more than 20 minutes. Around 40% of the failures lasted between 1 minute and 20 minutes and the rest were less than a minute. We also experimented with other recovery times.

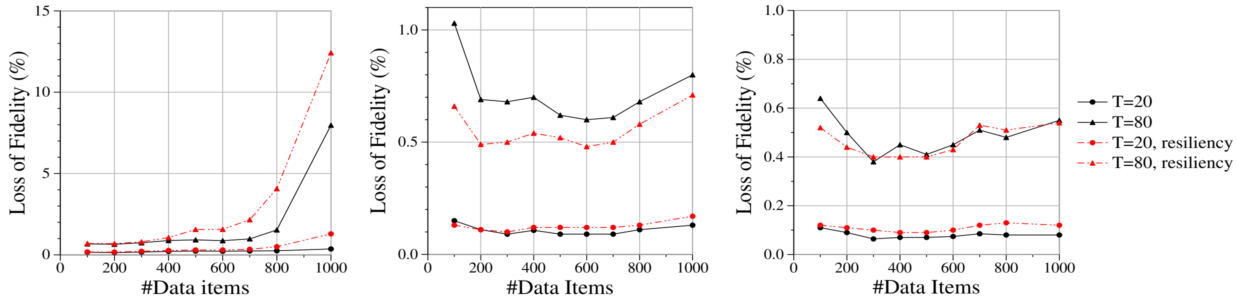
Between link failures and repository failures, the one that affects fidelity more is repository failure. A repository failure affects all its dependents whereas a link failure directly will typically affect only the dependents connected to that link. A link failure can be modeled as a partial failure of a repository - wherein the repository has failed only for some of its dependents. We have however not modeled such cases. We would like to mention here that though our failure model is incomplete our solutions are complete, they will work in the presence of repository failures or link failures.

7.4 Effect of Resiliency Features on Performance Loss

First we discuss fidelity loss in the absence of failures. Figure 7(a) shows the effect of adding resiliency to LeLA while Figure 7(b) shows the effect of adding resiliency to DiTA. In Figure 7(a) we see that the loss in fidelity due to addition of resiliency is negligible for number of data items < 400. However, beyond 800 data items, the loss in fidelity due to addition of resiliency increases considerably. As the number of data items increases, the cost incurred due to resiliency is much more than the benefits obtained (discussed next) and hence the increase in the loss in fidelity.

In Figure 7(b), in many cases, we observe that the resulting fidelity actually improves because of resiliency. This is because, if the back-up parent is not loaded then an update from the back-up parent might reach the dependent earlier than an update from the parent, hence improving the fidelity. We can see that any loss in fidelity due to addition of resiliency is less than 0.1%.

Given the superior performance of DiTA over LeLA both with and without resiliency, the rest of the results presented in this section are for DiTA.



(a) LeLA:Uniform Data Access (b) DiTA:Uniform Data Access (c) DiTA:Zipf Data Access

Figure 7: Effect of Resiliency on Fidelity – In the Absence of Failures

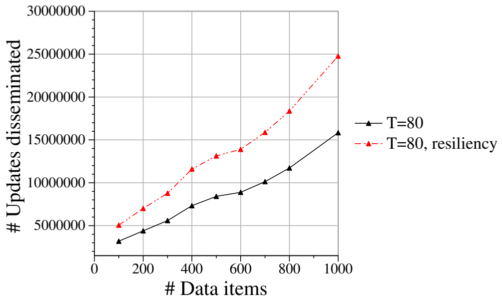


Figure 8: #Updates Disseminated in the Absence of Failures: Uniform Distribution.

Next, we see how DiTA performs when the data items a repository is interested in are chosen from a Zipf distribution. In Figure 7(c), we notice that the trend in the loss in fidelity is same as that in Figure 7(b). However, the loss in fidelity is less compared to that in the uniform access case due to popularity of some data items.

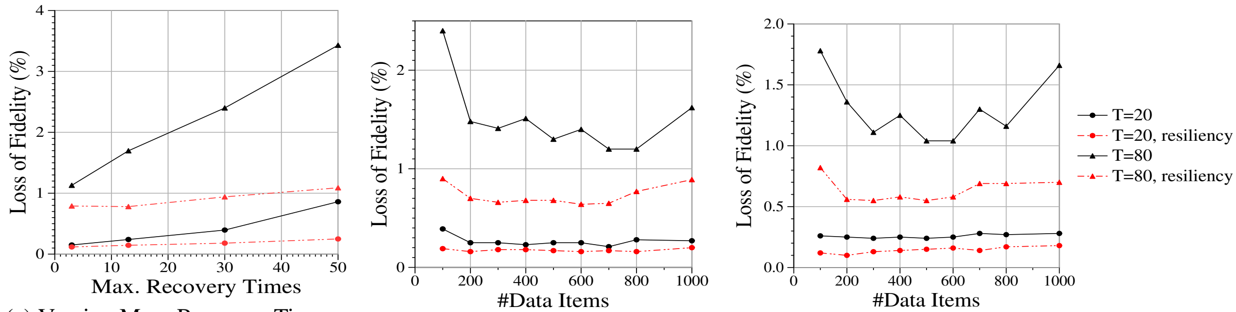
We wanted to understand the behavior in Figures 7(b) and (c) better. To this end we examined the number of updates disseminated by DiTA with and without resiliency. In Figure 8, we see that the number of updates disseminated in the network increases due to resiliency (in the absence of failures). Despite this increase in load, we see that the fidelity offered by 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 from the parent reached (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 those sent by the parent (since they see different views due to different coherence 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 a back-up parent is loaded, the updates sent by it will typically reach later than those sent by the parent. In this case, updates from the back-up parent are of no use to the dependent. This increases the loss in fidelity. As mentioned earlier, the dependent reduces this loss by discarding updates with time-stamps earlier than what it currently has for the same data item.

Finally Figure 9 presents the performance under failures. Figure 9(a) shows the effect of varying the maximum recovery time and Figures 9(b) and (c) show the effect of varying the number of data items. (The maximum recovery time for Figures 9(b) and (c) is 30 minutes.) These figures show that, (i) adding resiliency improves fidelity in failure situations, (ii) loss in fidelity for the Zipf data access is lower than the uniform data access (similar to the non-failure case) and (iii) with the addition of resiliency, the loss in fidelity is stable with respect to both increase in the number of data items and increase in the recovery times. Therefore, resilient DiTA displays good scalability properties.

8 Related Work

Firstly, it is important to point out the differences between the problem of resilient dynamic data dissemination and that of caching and replication.

1. Whereas a client may experience a *miss* when accessing a cache, with our dynamic data repositories a miss won't occur. Our work does employ the notion of cooperation as in "cooperative caching" but unlike in that scenario where a one-time use of a cached item in a cache occurs to satisfy the needs of a user of data present in another cache, here cooperation occurs to disseminate the data *continuously* from one repository to another.



(a) Varying Max. Recovery Times, for 100 data Items

(b) Uniform Data Access

(c) Zipf Data Access

Figure 9: Effect of Resiliency on Fidelity of DiTA – In the Presence of Failures

2. Turning to replication, whereas replicas are copies of each other, in our repositories, at any point in time, the value of the same data item at different repositories may be different.
3. Traditionally, caches and replicas have been employed to place needed data objects closer to the clients so as to minimize access latency and source load. In our approach, we employ cooperation to provide a data flow to clients and to avoid the load resulting from the source directly handling all the clients. Even though this is also akin to the goals of application-layer multicast, we are also concerned with achieving coherence, and hence, take communication and computational delays into account to improve fidelity.

We now turn our attention to push-based dissemination techniques that have been recently developed. These include broadcast disks publish/subscribe applications [17, 1], web-based push caching [11], and speculative dissemination [2].

The design of coherence mechanisms for web workloads has also received significant attention recently. Proposed techniques include strong and weak consistency [15] and the leases approach [7, 28]. Our contributions in this area lie in the definition of coherence in combination with fidelity requirements. Coherence maintenance has also been studied in the context of web caching [15] where hierarchical web proxy architectures [5] and cooperative web caching [27, 26, 28] have been proposed. 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 [13] where push-based invalidation and dependence graphs are employed to determine where and when to push invalidates. Achieving scalability by adjusting the coherence requirements of data items is studied in [29]. The difference between our approaches and theirs is that repositories don't cooperate with one another to maintain coherence.

Work on scalable and available replicated servers [29] is related to our goals. Whereas this work addresses the issue of adaptively varying the coherence requirement in replicated servers based on network load and application-specific requirements, we focus on adapting the dissemination tree for time-varying data.

Mechanisms for disseminating fast changing documents using multicast-based push has been studied in [23]. Here 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 coherence tolerances. Multicast tree construction algorithms in the context of application-level multicast have also been studied in the past [14]. 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 coherence of highly dynamic data, given specific data coherence requirement.

Turning to the caching of dynamic data, techniques discussed in [13] 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. Several research groups and startup companies have designed adaptive techniques for web workloads [4, 10]. 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 was studied in [20, 21]; the approach focuses on pushing individual data items directly to clients, based on client coherence 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 direct source-client based dissemination. Our results show that such cooperation reduces load the the sources and leads to lower loss of fidelity.

Our work can be seen as providing support for executing continuous queries over dynamically changing data [16, 6]. Continuous queries in the Conquer system [16] are tailored for heterogeneous data, rather than for real time data, and uses a disk-based database as its back end. NiagraCQ [6] 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).

9 Conclusions and Future Work

We examined the design of a data dissemination architecture for time-varying data. The architecture ensures data coherence, 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 coherence needs are explicitly taken into account.
- Design of a mechanism for making the cooperative dissemination network resilient to failures so that even under failures, data coherence is not completely lost. In fact, with our resiliency mechanisms, loss in fidelity decreases even under many non-failure situations. In failure situations, the mechanisms display attractive scalability properties.

In [25] we have also examined the problem of scheduling the various actions that a node takes based on coherence requirements of the dependents, the updated value and the cost and benefits of disseminating an update to the dependents. Whereas our approach uses push-based dissemination, other dissemination mechanisms such as pull, adaptive combinations of push and pull [3], as well as leases [19] could be used to disseminate data through our repository overlay network.

Also, using piggybacking will help us improve the fidelity further. Consider a repository P serving data items to a set of dependents $Q_1, \ldots, Q_n$. When P receives an update for a data item we can do the following. If a change is to be pushed to a dependent, we do not immediately send it but wait for a small time interval. If during that time interval P receives another update which is also of interest to the dependent, then the second update is piggybacked on to the first. All the updates are then pushed to the dependent at the end of the time interval in one single communication. In cases where there are no piggyback messages, addition in the delay at the node may increase the loss in fidelity. Hence, it is important that we piggyback only when needed.

The use of such efficiency improvement mechanisms as well as the evaluation of our mechanisms in a real network setting is the subject of future research.

Acknowledgments

We would like to thank the reviewers for their very useful comments. We would also like to thank to Allister Bernard and Vivek Sharma for their contributions in the initial stages of this work and Tata Consultancy Services for sponsoring our laboratory for internet research.

References

[1] 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.

[2] 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.

[3] 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*, 2002.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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

[10] 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.*

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

[12] G. Iannaccone and C. Chuah and R. Mortier and S. Bhattacharya and C Diot. Analysis of link failures in an IP backbone. In *Internet Measurement Workshop*, 2002.

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

[14] D. Li and D. Cheriton. Scalable Web Caching of Frequently Updated Objects Using Reliable Multicast. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

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

[16] 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.

[17] 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.

[18] *http://www.openclinical.org/aispi_neoganesh.html*.

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

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

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

[22] 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.

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

[24] S. Shah, K. Ramamritham and P. Shenoy, Maintaining Coherency of Dynamic Data in Cooperating Repositories, *Proceedings of the 28th Conference on VLDB, August 2002*.

[25] S. Shah, S. Dharmarajan and K. Ramamritham, An Efficient and Resilient Approach to Filtering and Disseminating Dynamic Data, *Proceedings of the 29th Conference on VLDB, September 2003*.

[26] 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.

[27] 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.

[28] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Hierarchical cache consistency in a WAN. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

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