Maintaining Temporal Coherency of Cooperating Dynamic Data Repositories

Shetal Shah Allister Bernard Vivek Sharma Krithi Ramamritham Prashant Shenoy†
Department of Computer Science and Engineering, †Dept of Computer Science,
Indian Institute of Technology Bombay, University of Massachusetts,
Mumbai, India 400076. Amherst, MA 01003, USA.

Abstract

On-line decision making often involves significant amount of time-varying data. Examples of time-varying data include financial information such as stock prices and currency exchange rates, real-time traffic and weather information, and data from industrial process control applications. The coherency requirements associated with time-varying data depends on the nature of the data and user tolerances. This paper examines techniques to efficiently disseminate time-varying data from sources to a set of repositories. A particular focus of our work is to examine how such repositories can *cooperate with one another and the source* to improve the efficiency of the dissemination process, while meeting user coherency requirements. We consider two key issues: (i) When should the source and/or the repositories push changes of interest to other repositories so as to meet all user-specified coherency requirements? (ii) How should an overlay network of such repositories be organized so as to minimize the overheads of maintaining temporal coherency of all data items stored in the various repositories? We examine these questions in turn, offer a set of alternative solutions to address these questions and experimentally evaluate their performance using real-world traces of dynamically changing data (specifically, stock prices). We show that cooperation helps reduce the system-wide overheads for maintaining coherency across all repositories. However, contrary to intuition, we also show that increasing the degree of cooperation beyond a certain point can, in fact, be *detrimental* to the overall goals of achieving high fidelity at low overheads. To address this issue, we propose techniques to (i) derive the "optimal" degree of cooperation among repositories, and (ii) derive the logical structure of an overlay network of cooperating repositories so as to maintain temporal coherency of data at low cost.

Keywords: Dynamic Data, Temporal Coherency, Data Dissemination, Cooperating Caches

1 Introduction

On-line decision making often involves significant amount of time-varying data. Examples of time-varying data include financial information such as stock prices and currency exchange rates, real-time traffic and weather information, and data from industrial process control applications. The coherency requirements associated with a time-varying data item depends on the nature of the item and user tolerances. To illustrate, a casual observer of currency exchange rate fluctuations may be willing to tolerate some incoherency, whereas a user involved in exploiting exchange disparities in different markets will require a stronger coherency requirement. Similarly, a streaming stock ticker application can use data with a lower coherency (e.g., stock prices delayed by 20 minutes) than online stock trading that imposes stringent coherency requirements (e.g., real-time stock prices).

Sources of time-varying data can often become a bottleneck, especially when serving a large number of clients. One technique to alleviate this bottleneck is to replicate data across multiple repositories and have clients access one of these repositories. Although such replication can reduce load on the sources, it introduces new challenges—unless data are carefully disseminated from sources to repositories, either (a) data in the repositories will violate user coherency requirements, or (b) the overheads involved in such dissemination will be substantially larger than is necessary to optimally meet user coherency requirements.

This paper examines techniques to efficiently disseminate time-varying data from sources to a set of repositories. A particular focus of our work is to examine how such repositories can *cooperate with one another and the source* to improve the efficiency of the dissemination process, while meeting user coherency requirements. To address this problem, we consider an architecture wherein a source *pushes* changes that exceed specified thresholds to repositories connected to it, a repository cooperates with other repositories, again, by pushing data of interest to them, and also serves users directly connected to it. Given such an architecture, we consider two key issues:

1. When should the source and/or the repositories push changes of interest to other repositories so as to meet all user-specified coherency requirements?
2. How should an overlay network of such repositories be organized so as to minimize the overheads of maintaining temporal coherency of all data items stored in the various repositories?

In the rest of this section, we first define the problem of maintaining temporal coherency for a data item and then describe the challenges in doing so in a network of cooperating repositories.

1.1 Temporal Coherency Semantics

Consider a user interested in time-varying data items. Assume that the user obtains these items from a data repository instead of the source server. In such a scenario, the repository must track dynamically changing data so as to provide users with temporally coherent information. Assume that the user specifies a temporal coherence requirement (tcr) for each item of interest. The value of the tcr denotes the maximum permissible deviation from the value at the source, and thus, constitutes the user-specified tolerance. The tcr can be specified in units of *time* (e.g., the item should never be out-of-sync by more than 5 minutes) or *value* (e.g., a stock price should never be out-of-sync by more than a dollar). In this paper, we only consider temporal coherence requirements specified in terms of the value of the object (maintaining temporal coherence specified in units of time is a simpler problem that requires less sophisticated techniques). To maintain coherence, each data item in the repository must be refreshed in such a way that the user-specified coherency requirements are maintained. Formally, let $S(t)$ and $U(t)$ denote the value of the data item at the server and the user, respectively, at time t (see Figure 1). Then, to maintain temporal coherence , we should have $|U(t) - S(t)| \leq tcr$.

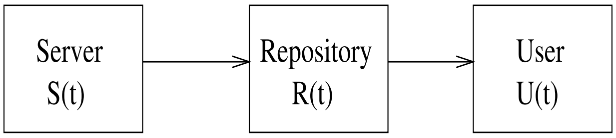


Figure 1: The Problem of Temporal Coherence

Although Figure 1 shows a single data repository, the temporal coherency requirements are no different if there are multiple data repositories acting as intermediaries between the server and the end-user. The design of such a cooperating data repository architecture and associated algorithms for reducing the computational and communication overheads incurred in maintaining temporal coherence is the focus of this paper.

The effectiveness of such a cooperating repository can be quantified using a metric referred to as *fidelity*. The fidelity of a data item is the degree to which a user's temporal coherence needs are met. We define the fidelity f observed by a user to be the total length of time that the above inequality holds (normalized by the total length of the observations). The goal of a good coherency maintenance technique is to provide high fidelity at low cost.

1.2 Maintaining Temporal Coherency in an Overlay Network of Cooperating Repositories

A cooperative repository architecture consists of one or more sources, multiple cooperating repositories and several clients (see Figure 2). Each client connects to one of the repositories for accessing data items. For each data item of interest, a coherency requirement tcr is specified to the repository. A repository caches data items of interest to its users and maintains coherency of the cached data in cooperation with the source and other repositories. By cooperation, we mean, upon refreshing a data item, the repository pushes the refreshed value to other repositories interested in that item (thereby alleviating the source from the burden of pushing these updates to all interested repositories). As shown in Figure 2, the repositories are logically connected to form the *repository overlay network*, abbreviated as *ron*. Updates to data items are disseminated through this overlay network so as to maintain user-specified coherency requirements at each repository. Observe that without such cooperation, each repository would need to refresh cached data directly from the source, which increases the load on sources and the client response times and limits scalability.

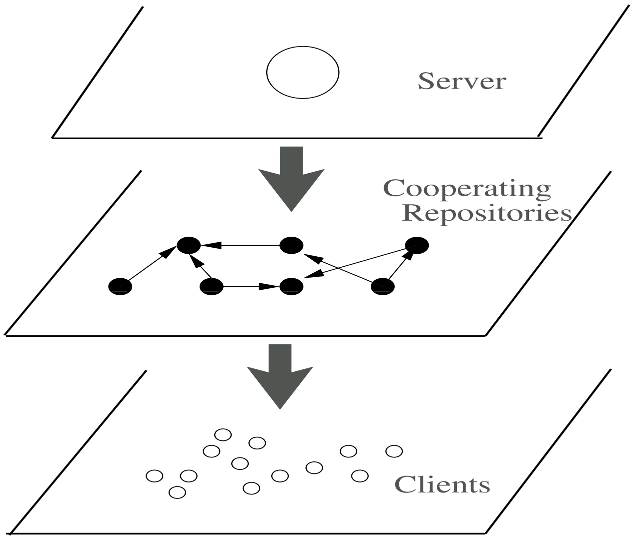


Figure 2: The Cooperative Repository Architecture

In this paper, we examine the problem of how such an overlay network of cooperating repositories can provide high fidelity for time-varying data. Specifically, we examine techniques to (a) reduce the end-to-end delays in propagating these updates to repositories in an overlay network, and (b) ensure that all interesting updates to data items are disseminated.

Achieving these goals requires answers to the following interrelated issues:

1. When should a repository disseminate updates (that it receives) to other repositories connected to it?

   Observe that different users accessing the same data items can have different coherence needs. The coherency requirement of a data item in a repository is determined by the most stringent tcr across all interested users. In the presence of cooperation, each repository caters to the needs of other repositories (in addition to servicing its users). Since different repositories can have different coherence needs (by virtue of the different tcrs specified by their users), a repository will need to take these differences into account when disseminating updates to its neighbors.

2. How much cooperation should a repository offer to the rest of the network?

   Given that repositories cooperate with one another, a repository may hold data beyond what its own users may need. We show that this altruism pays off in reducing the system-wide overheads for maintaining coherency across all repositories. However, contrary to intuition, we also show that increasing the amount of cooperation beyond a certain point can, in fact, be *detrimental* to the overall goals of achieving high fidelity at low overheads. To address this issue, we propose techniques to derive the "optimal" degree of cooperation among repositories.

3. What should the (logical) interconnection between the repositories be, i.e., who serves whom and what?

   The structure of the ron determines (i) the maximum end-to-end delay for disseminating updates (which depends on the repository farthest from the source) and (ii) the overhead of disseminating updates at each repository (which depends on the number of dependents served by the repository). The ron should be structured so as to balance these tradeoffs. We develop two algorithms to answer this question. 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 carefully chosen path to make this decision. The performance of these approaches is compared to an idealized approach that attempts to minimize the length of the path from the source to all repositories.

In the following sections, we examine each question in turn, offer a set of alternative solutions to address these questions, and experimentally evaluate their performance using real-world traces of dynamically changing data (specifically, stock prices).

The rest of the paper is structured as follows The next three sections offer solutions to the three problems mentioned above: Algorithms for disseminating dynamic data from one repository to another (or from a source to its clients) so that temporal coherency of data is maintained are discussed in Section 2. Determining the level of cooperation between repositories is the subject of Section 3. Algorithms for placing a repository in an existing repository structure are discussed in Section 4. Results from an extensive evaluation of our algorithms using real-world traces of dynamic data are presented in Section 5. Related work is discussed in Section 6. Section 7 concludes the paper with a summary and directions for future work.

2 Data Dissemination to Maintain Temporal Coherency of Repositories

In this section, we show how to set the coherency requirements of a data item d served by repository P to a repository Q so that the coherency needs of the latter are satisfied.

Consider a source S that disseminates data item d to a repository P, which in turn disseminates d to repository Q. Henceforth, we refer to Q as a *dependent* of P.

Let c^p and c^q denote the coherency requirements of data item d at repositories P and Q, respectively. Since P serves Q,

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

Thus, to effectively disseminate updates, we require that the coherency requirement at a repository should be *at least as stringent as those of its dependents*.

Let u_i^s, u_{i+1}^s, u_{i+2}^s ... denote a sequence of updates to d at the source S. Let u_j^p, u_{j+1}^p, u_{j+2}^p ... denotes the updates received by P and u_k^q, u_{k+1}^q, u_{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 u_j^p received by P is forwarded to Q if

$$\mid u_j^p - u_k^q \mid \quad \geq \quad c^q \tag{2}$$

where u_k^q denotes the previous update received by Q. Intuitively, Equation 2 indicates that any update that violates the coherency requirements of Q is forwarded to Q.

Note that this is a necessary but not sufficient condition for maintaining temporal coherency at Q. To understand why, suppose u_i^s, u_j^p and u_k^q represent the value of d at S, P and Q, respectively. Let the next update at the S be u_{i+1}^s such that

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

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

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, a dependent of P, will also miss this update (causing a violation of its coherency requirement). Figure 3 provides an example of this situation. Thus, even under ideal conditions of zero processing and link delays, a dissemination technique that solely uses Equation 2 to disseminate updates might not provide 100% fidelity (indicating Eq. 2 is not a sufficient condition to maintain coherency). Hence, dissemination algorithms need to be developed carefully to avoid such problems (i.e., should ensure that a repository does not miss any updates of interest to itself or its dependents).

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

2.1 Data Differencing

The “missed updates” problem described earlier occurs when an update u_{i+1}^s, where $u_i^s < u_{i+1}^s < u_i^s + c^p$, satisfies both Equations 3 and 4.

From these equations, we get,

$$\mid u_{i+1}^s - u_j^p \mid - \mid u_{i+1}^s - u_k^q \mid \quad < \quad c^p - c^q \tag{5}$$

which reduces to

$$c^q \quad - \quad \mid u_j^p - u_k^q \mid \quad < \quad c^p \tag{6}$$

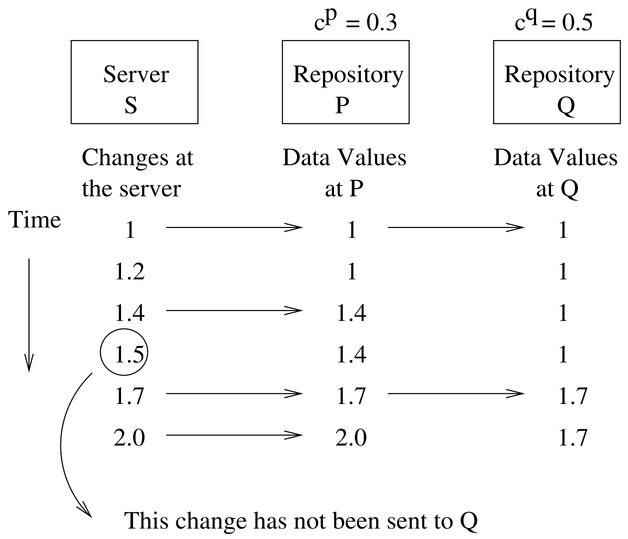


Figure 3: An example illustrating the need for careful dissemination of changes.

See Appendix A for a proof of the derivation of Equation 6. Equation 6 represents the additional condition that needs to be checked to see if an update should be disseminated to a dependent. Thus, the dissemination technique propagates an update u_j^p received by P to dependent Q if either Equation 2 or 6 is satisfied. In the example illustrated in Figure 3, such a technique would propagate the update corresponding to value 1.4 from P to Q (since it satisfies Equation 6). Consequently, the subsequent increase in value to 1.5 does not result in a tcr violation at Q. Note that the update of 1.4 is not strictly required as per the coherency requirement of Q (Equation 2), but is essential to prevent the "missed updates" problem.

Such a dissemination technique incurs the following overheads:

- *Computational Overhead*: The computational overhead arises from verifying Equations 2 and 6 for each dependent when a data update is received. Observe that, at the very least, upon receiving an update, a naive approach would incur the overhead of verifying Equation 2. The additional burden of verifying Equation 6 imposes a marginal additional overhead on a repository.

- *Network Overhead*: Since a repository must receive the updates necessary to maintain coherency of its dependents' data, in addition to the updates needed to maintain the tcr of its users, the network will need to carry additional messages. This is because Equation 6 as well as Equation 4 need to be satisfied.

- *Space Overhead*: This algorithm incurs *no* space overhead since no additional (state) information needs to be maintained for verifying Equation 6 other than what is already maintained at the repository.

2.2 Coherence Reassignment

We now examine a variant of data differencing, wherein instead of checking Equation 6 upon receiving each update, the coherency requirements of the dependent are changed permanently so as to subsume Equation 6. Rewriting Equation 6 we get

$$| \; u_j^p - u_k^q \; | \quad > \quad c^q - c^p \tag{7}$$

An implication of this equation is that a repository should also forward updates that exceed $c^q - c^p$. Since $c^q - c^p$ is less than the original coherence requirement of Q, using this as the new coherency requirement ensures that Q never misses any update of interest.

Observe that, as per Equation 1, we require that the coherency requirement of a parent must be more stringent than its dependents. In the event that $c^q - c^p$ is smaller than c^p, this condition is no longer true, indicating that P can no longer service Q. Such a scenario can be handled by simply reassigning the coherency requirement of Q to be same as P. Since this is the only difference from the data differencing approach, this approach incurs the same network and space overheads as data differencing.

2.3 Unique Coherence-based Dissemination

In this approach, the source maintains a list of all unique coherency requirements for a data item d specified by various repositories. For each such coherency requirement, the source also tracks the last update disseminated for that coherency requirement. Upon a new update, the source examines each unique coherency requirement and the last update sent for that tcr. It then determines all tcrs that are violated by the update. The update is tagged by the maximum such coherency requirement c_max and the tagged update is then disseminated through the overlay network. The source also records this data value as the last update sent for all tcrs that are less then 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 coherency requirement less than or equal to the tagged value.

It can be shown that this dissemination algorithm achieves a fidelity of 100% (in the absence of network transmission delays). To see why, consider a data item d with a value of v. Let the coherency requirement at repository P be c. Suppose that an update causes the value of d 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 coherency requirement has been violated for c. Hence, an update with the new value v' is tagged with $c' \geq c$ and disseminated through the overlay network. Consider the path from the source to repository P. As per Equation 1, the coherency 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 coherency tolerance, 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 network delays.

We now discuss the overheads of this approach.

- Computational Overhead: The computational overhead involves finding the maximum coherency value, if any, affected by the an update at the source. A large network of cooperating repositories can result in a large overhead (especially if the list of unique tcr values is also large). To avoid scalability problems, this computation can be migrated (i.e., moved) from the source to the repositories that are served directly by the source. The source then sends *every* update to them and the first-level repositories in turn execute the above algorithm to decide which updates to propagate further and with what tag. Thus, the overhead can be distributed across multiple repositories, which alleviates the burden on the source.

- Network Overhead: Since this approach disseminates updates only when necessary and only to repositories that need the update, the approach makes efficient use of network resources. However, if the approach is augmented using the computation migration technique suggested above, then the source will disseminate more updates than absolutely necessary to the first-level repositories; other repositories continue to receive updates only when necessary.

- Space Overhead: The algorithm imposes a state space overhead at the source (or at the first-level repositories) to store the list of all unique coherency tolerances associated with each data-item and the last update sent for each tcr.

We have experimented with all three data dissemination approaches for maintaining temporal coherency and report on the effects of these overheads on fidelity in section 5.

3 How Much Should a Repository Cooperate?

In the previous section, we assumed that the topology of the ron was known *a priori* and examined the issue of how updates should be disseminated by repositories to their dependents. In this section and the next, we consider the issue of how to construct the ron. We define the degree of cooperation to be the number of dependents served by a repository. A high degree of cooperation allows a repository to take on increased responsibilities, which reduces source overload and improves fidelity. On the other hand, a repository with a high degree of cooperation can also indirectly lead to a loss in fidelity (since this increases the computational overheads at a repository, which could become a bottleneck). Hence, it is important to determine a degree of cooperation that balances various overheads.

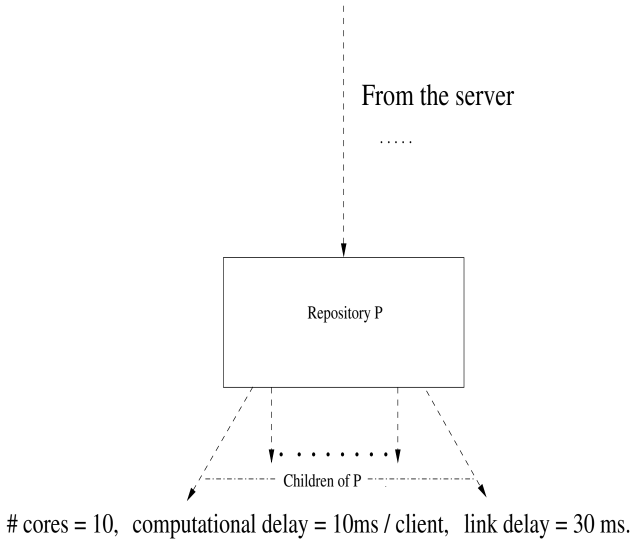


Figure 4: An Example Illustrating the Need for Careful Cooperation

To achieve this objective, let us first understand the overheads involved in disseminating updates. Data dissemination overheads include two components:

1. *Network delays:* This is the network delay incurred in propagating an update from a repository to a dependent. It includes the delays on all physical links from a repository to its dependent.

2. *Processing delays:* This is the delay resulting from the computations performed by a repository to determine whether an incoming data change is to be forwarded to one of its dependents.

The smaller these delays are, the higher the fidelity of the system.

Let us now consider the tradeoff between network delays and processing delays. To better illustrate this tradeoff, consider an example where the ron is a tree (see Figure 4). Let repository P be an intermediate node in the tree and let P have 10 resources (push connections) to offer towards the cooperative effort. This implies that P can handle up to 10 dependents. Assume that it takes 10 ms to push a data change to a dependent. If all the resources of P are used up, i.e., it has 10 dependents, then the worst-case processing delay experienced by any dependent of P is around 100 ms. This implies more time is spent processing a data item than is spent communicating it to a dependent. In other words, in this system, the computational delays dominate the network delays. If we restrict the number of dependents of P to a number smaller than 10, say 2, then the worst-case processing delay will reduce to 20 ms, but the cooperative network will have a larger depth. Thus, the example illustrates that a greater degree of cooperation increases the processing delay but reduces the end-to-end network delay (by the virtue of reducing the number of hops from the source to the farthest repository). On the other hand, a small degree of cooperation reduces the processing delay at a node but increases the end-to-end network delays. In the extreme case, if the degree of cooperation is reduced to one, the ron becomes a linear chain of repositories with a large network delay. To maximize fidelity, the ron should be constructed such that the *sum of two delay components is minimized*.

For a given set of repositories, the variation in (loss of) fidelity with increasing degree of cooperation exhibits a U-shaped curve (see Figure 5). The point where the loss in fidelity is minimized depends on the network and processing delays incurred by the ron. In the falling part of the curve, the network delays dominate and in the rising part, the processing delays dominate. The figure shows that *arbitrarily increasing the degree of cooperation can, in fact, be detrimental to fidelity.* Hence, in a system where link delays dominate, it is prudent to use a high degree of cooperation. On the other hand, if processing delays dominate, then a small degree of cooperation should be chosen (i.e., each repository should have a small number of dependents). In other words, the degree of cooperation should be directly proportional to the network delays and inversely proportional to the processing delays. This results in the

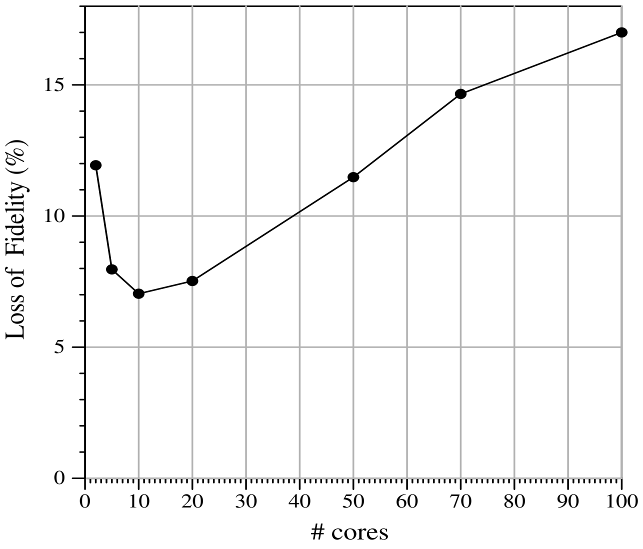


Figure 5: Variation in Loss of Fidelity with Offered Cooperative Resources

following heuristic to compute the actual number of resources to be used for cooperation:

$$actual\#cores = \min(\frac{net\_delay}{comp\_delay \times C}, \#cores) \tag{8}$$

where $\#cores$ denotes the upper bound on the degree of cooperation (determined by the resources available at a repository), net_delay and $comp\_delay$ denote the network and processing delays, respectively, and C is a constant that indicates the percentage of dependents typically interested in an update. The C factor is used to calculate the computational delays expected at a node. The above formula allows us to set the level of cooperation depending on the link and expected computational delays. In Section 5 we study the effect of adaptively determining the degree of cooperation and compare it to a scenario where each node uses all its resources for cooperation.

In the rest of the paper, we refer to the *number* of *co*operative resource$s$ of a repository *actually used towards cooperation*, by actual#cores. $actual\#cores$ reflects the actual degree of cooperation and is bounded by $\#cores$. The ron should be structured to respect the constraint specified by $actual\#cores$.

4 Determining the Logical Structure of Cooperating Repositories

Thus far, we have examined how each repository should disseminate updates to its dependents and to what degree the repositories should cooperate with one another. In this section, we examine how to construct the logical structure of the ron. Specifically, we propose three alternative algorithms for constructing the ron. For ease of exposition, we examine the problem assuming a single source for all data items; our algorithms can be extended to handle multiple sources.

The first of these algorithms, the *Shortest Path Algorithm (SoPA),* constructs the ron by assuming global knowledge of all repositories and creates an overlay network such that the total path length from the source to the repositories is minimized. In doing so, the algorithm takes the underlying physical structure of the network into account and builds a shortest path tree so as to avoid retransmission of messages over a common subpath between a source and multiple repositories. While this algorithm is not likely to be practical because of its centralized nature, we offer this algorithm as a baseline against which we compare the overheads and performance of the more practical algorithms.

Our other two algorithms are *incremental* in nature and allow the ron to be constructed one repository at a time (by allowing dynamic insertions and deletions of repositories). In these algorithms, a repository joins the ron by sending an insert message to the source of the data item of interest. The algorithm then evaluates whether the source can directly disseminate data to the new repository. If not, the algorithm attempts to *insert* the new repository in the existing ron by

- finding a position in the ron for the new repository, and
- finding suitable parents to satisfy the needs of the new repository.

Our *Best Path based Algorithm (BePA)* achieves these objectives by computing a preference factor for each repository and uses it to determine the position of a new repository. The algorithm attempts to position preferred repositories closer to the source. When a new repository is inserted in the ron, instead of searching through the entire ron, BePA chooses the new repository's position by searching along a carefully selected route. BePA takes both link delays and computational delays into account and uses the tcr values of repositories' data items but does not take the rate of change of each data item into account.

In the *Level by Level Algorithm (LeLA)*, repositories are organized in levels, with the ron forming a dag such that edges of the graph go only from one level to the next. A node in the graph represents a repository and an edge from repository P to repository Q implies that P provides data to Q. Each repository in a level, say l, will have parents in level $l-1$ and dependents in level $l+1$. The idea behind this approach is that starting from the source of data, each level is examined for its suitability to absorb a new repository. The decision is left to the "leader of a level" called the *load controller*. Each load controller is connected to its adjacent load controllers. The function of the load controller at a level is to find suitable parents for the new repository, if that level is found to be suitable for the new repository.

All the algorithms construct a dag, if repository Q has an edge from P from it, this is interpreted as P having the potential to serve Q. Note that in general, a repository may have multiple dependents and may depend on multiple repositories. All algorithms also ensure that the coherency requirements of a repository P are more stringent than those of any dependent Q (i.e., $c^p \leq c^q$). Unlike BePA, which considers only the repositories along a particular path, LeLA takes into consideration the characteristics of the repositories in the ron, level by level. However, unlike SoPA, which does this directly and in a centralized fashion, LeLA achieves the effect indirectly via its load controllers and in an incremental manner. The latter property also applies to BePA.

In the rest of this section, we provide details of these algorithms.

4.1 SoPA: Shortest Path Based Approach

Figure 6 provides the intuition and motivation underlying SoPA. The route from the server to repository P has some links in common with the route to another repository Q. Along those common links, a data update will be sent twice. Consequently, instead of creating individual data flows from the server to each repository, it would be better to create a ron that capitalizes on the network topology. Towards this end, SoPA creates a ron which corresponds to a tree, such that the load on the common links and the source is reduced, potentially at the cost of an increase in the delay in disseminating updates to Q (see Figure 6). Observe that, the tree construction process should ensure that such delays do not affect the fidelity of disseminated data. To do so, SoPA ensures that the constraint specified by ($\#cores$) of each repository is satisfied, which in turn reduces the transmission delays from the source to the repositories.

The network of repositories can be thought of as a graph with the source and the repositories being the nodes, the paths between the repositories being the edges, and edge weights being the transmission delays between two repositories. The problem then is to find a shortest path tree in this graph, rooted at the source, that spans all repository nodes and is constrained by repositories' $\#cores$ (i.e., limits the number of dependents of each repository to a value no greater than its $\#cores$). The details of such as shortest path algorithm are as follows.

1. Use Dijkstra's algorithm [10] to create a shortest path tree rooted as the source with the link delays represented by the edge weights.

2. After a shortest path tree has been generated using the Dijkstra's algorithm, we check the tree for repositories with more dependents than their corresponding $\#cores$. For each of these repositories, we prune the subtree rooted at that repository by removing its smallest subtrees, where the number of removed subtrees is (outdegree - $\#cores$). This leaves the repository with $\#cores$ dependents. Nodes in the removed subtrees are placed in the set $V-S$. (Dijkstra's algorithm keeps two sets of vertices: S - the set of vertices whose shortest paths from the source have already been determined and $V-S$ - the remaining vertices, where V is the set of vertices in the graph.) Any repository whose $\#cores$ was not satisfied is similarly placed in $V-S$.

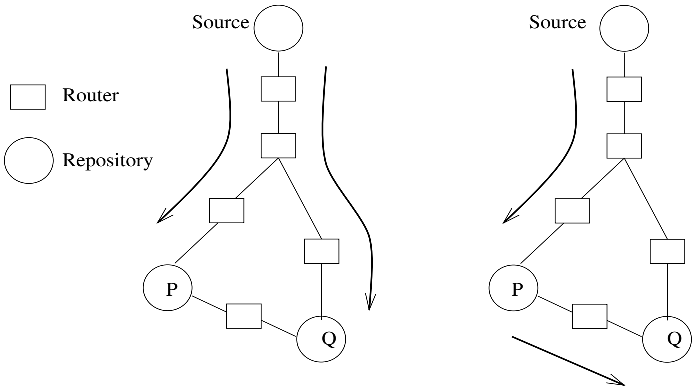


Figure 6: An example to illustrate the benefits of SoPA.

3. If no repository is found to violate its $\#cores$, go to step 4, else recursively run steps 1 and 2 on the set $(S, V-S)$ again.
4. After the tree has been constructed, the coherency of a data item in a repository is determined. For a data item d in repository P, this is set to be the strongest coherency for d required by any repository in the entire subtree rooted at P.

Insertion of a new repository or the deletion of an existing repository will trigger a recomputation of the ron according to the algorithm above. Note that, a change in the coherency requirements of a data item or a change in the data set of a repository *does not* require a recomputation of the ron. Rather, only the tcr values of the parent and the ancestors need to be recomputed as per Step 4 of the algorithm.

4.2 BePA: Best Path based Algorithm

BePA attempts to create the ron by taking into account computational delays, link delays, and data items held by repositories. These properties of a repository are encoded using a single *preference* factor which reflects the degree to which a repository can serve high fidelity data to its dependents. BePA attempts to insert preferred repositories closer to the source. To insert a new repository, the algorithm selects an *insertion* path to follow within the ron. This path is made up of repositories $P_1, P_2...P_n$ where P_1 is the root(source) and P_i represents a repository at the i^{th} stage of the search. The next position P_{i+1} along the path is selected by examining the dependents of the current repository P_i and a decision is made whether to search further along the path or insert the repository at the current position P_i. The steps involved in satisfying an insert request are outlined in Section 4.2.2. First, we examine how the preference factor of a repository is assigned.

4.2.1 Computing a Repository's Preference Factor

The preference factor of a repository serving data in a set D is given by

$$pref_i^s = \frac{\sum_{d \in D}(c_d \times \#cores)}{|D|} \tag{9}$$

where c_d is the tcr value of data item d served by this repository, $\#cores$ is maximum number of users that can require d and $|D|$ is the number of data items that the repository needs. The lower the preference factor of a repository, the closer it should be to the source. This is because a small preference factor indicates that a repository has one or more of the following properties:

- Has less load since it is serving fewer users,
- Has the ability to serve a larger set of data items,

- Can meet more stringent coherency requirements.

Equation 9 represents an *intrinsic* property of the repository, which is independent of the network environment and depends only on its users and their coherency tolerances. In addition, we define the following *extrinsic* property for a repository.

$$pref_i^l = \frac{\sum_{k=1}^{m} l_{d_k}}{m} \tag{10}$$

where l_{d_k} represents the link delay between repository i and its dependent k in the cooperative network. Observe that $pref_i^l$ depends solely on the network environment and is independent of the user needs at a repository.

Using equations 9 and 10 we can calculate the overall preference for a repository vis-a-vis its position in the ron.

$$pref_i = pref_i^s \times pref_i^l \tag{11}$$

4.2.2 Details of BePA

Let the new repository being inserted be P_{ins}.

1. The algorithm starts from the source $S = P_1$; let us define $pref_1 = 0$. Then the next repository P_{i+1} along the *insertion* path is selected from the dependents of the *current repository* P_i such that P_{i+1} is the dependent with largest data item similarity with the inserted node P_i and has resources available. If not, P_{i+1} is the dependent with the largest preference factor.

2. An *insertion* path is followed until either the current repository P_i is less preferable than the repository being inserted P_{ins} or the current repository P_i has resources to serve the repository being inserted P_{ins}.

   (a) If P_i is less preferable than P_{ins} then the algorithm swaps these two repositories. In other words, P_{ins} is inserted in the position formerly occupied by P_i and P_i becomes the child of P_{ins}. Additionally, P_{ins} takes on as many dependents of P_i as possible (depending on resource availability); the remaining dependents continue to be served by P_i. Finally, the set of data items served by P_{ins} is computed as the union of the data items required by all its dependents.

   (b) Else if the current repository P_i is preferable to P_{ins} and has resources to serve the inserted repository, then P_{ins} is inserted as the child of P_i.

3. Once the repository is inserted it may find that not all of the data items it needs can be provided by its current set of parents. So it looks for additional parent repositories to provide these data items. If no such repositories exist then the existing parents are augmented to provide these data items. For this, BePA considers repositories within a maximum number of *hop*s between itself and the source.

4.3 LeLA: Organizing the Repositories Level by Level

As mentioned earlier, the idea behind this approach is that starting from the source of data, the load on repositories at different levels is examined for their suitability to absorb a new repository. This decision is made by the load controller at each level.

For each data item d, four parameters influence the load of a repository: (a) c_d, the coherency requirement, (b) r_d, the rate of change of data item d, (c) p, the processing speed of the repository, namely, the number of dependents it can disseminate data to, per unit time. The load of a repository can be calculated as follows: The time spent on processing data item d is $\frac{\#cores \times r_d}{p \times c_d}$. This is because r_d/c_d is roughly the number of changes in data item d per second. For each such change the processor deals with up to $\#cores$ repositories and for each dependent the time taken is $1/p$. Hence the total static load on a repository P is

$$load(P) = \frac{1}{p} \times \sum_{d \in D} \frac{\#cores \times r_d}{c_d} \tag{12}$$

When a repository P wishes to enter the network it sends a request to the source with the list of data items of interest, their tcr values, and its $\#cores$. The source examines if it can place the repository at the first level, if not it passes it on to the load-controller of the next level.

The function of the load controller at a level is to find suitable parents for the new repository, if that level is found to be suitable for the new repository. The parameters which govern the selection of a parent are:

1. Parent's load, which determines the computational delays incurred at a parent to disseminate a data change to its dependents.

2. Distance of the new repository from the server, which is the sum of the distance of the parent from the server and distance of the new repository from the parent. This determines the communication delays incurred by a data update.

3. The number of data items that a parent can provide to a repository at the given coherency without augmentation. This will help reduce the extra work that may be entailed at a parent if it had to augment its current commitments.

To achieve the above effect, upon receipt of an insert request the load controller at level i does the following: For each repository Q, at its level, which still has resources available, it calculates a preference factor as $load(Q) \times dist(P, Q) + \min dist(Q, server)$ (again, the lower the preference value of a repository, the more preferred the repository is). Among the more preferred repositories, the repositories which can satisfy a maximum of P's data requirements are made the parents of P. Some items that P requires may not be currently served by the parents. In such a case, these items are assigned to the most preferred parent.

5 Experiments and Results

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

5.1 Experimental Methodology

Our experiments model the underlying physical network using a randomly generated topology consisting of nodes (routers and repositories) and links. One of the nodes is selected as the source. The routing tables of all the nodes are generated using an all-pairs shortest path algorithm (by Floyd and Warshall [10]). After generating the physical network topology, we then generate the topology of the repository overlay network (ron). This is done using one of the three techniques discussed in Section 4. Specifically, SoPA examines all repositories in the network at once to generate the ron, while BePA and LeLA do so in an incremental manner. In the latter techniques, repositories are inserted into the ron one at a time. Each repository is assumed to have a data set associated with it consisting of (data item, tcr) pairs.

After generating the topology of the ron, the simulation of the data dissemination algorithms discussed in Section 2 is started. The performance characteristics of these algorithms 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) of a few companies obtained from *http://finance.yahoo.com*. The traces were collected at a rate of 2 or 3 stock quotes per second. Since the rate of change of any stock quote is much greater than even 1 change per second, the traces can be considered to be "real-time" traces. The details of the traces are listed in the table below. Specifically, upon each update to the stock price, the source determines whether to forward the update to the first-level repositories in the ron; each repository receiving the update then decides which of its dependents to forward the update. These dissemination decisions are made using either data differencing, coherence reassignment or unique coherency-based techniques.

Company	Date	Time	#Items	Max Value	Min Value
Microsoft	June 8, 2000	21:02-23:48 hours	7078	65.875	64.625
Veritas	June 8, 2000	21:20-23:48 hours	10000	135.75	131.50
DELL	June 1, 2000	21:56-22:53 hours	6852	43.75	42.87
CBUK	June 2, 2000	18:31-21:57 hours	10000	8.625	8.25
INTC	June 2, 2000	22:14-01:42 hours	10000	134.50	132.50

Traces used for the experiment

For our experiments, we vary the size of the physical network from 400 nodes to 1000 nodes. Unless specified otherwise, we present results only the 400 node scenario (1 source, 100 repositories and 300 routers). Results for other network sizes are briefly discussed in Section 5.3.5.

Each repository requests a randomly chosen subset of the data items stored at the server.

Our experiments assume a network (link) delay of 25 milliseconds and a computational delay of 25 milliseconds per dependent for each update. Results of experiments involving other link and computations delays are presented in Section 5.3.2.

We use different mixes of coherency tolerances for our experiments. Specifically, the tcr on the stock prices consist of a mix of stringent tolerances (varying from \$0.01 to 0.1) and less stringent tolerances (varying from \$0.1 to 0.9). We use a uniform distribution T to decide what fraction of the data items have stringent coherency requirements at each repository (the remaining fraction, $1-T$, of data items have loose coherency requirements). The parameter $\#cores$—the resources offered by each repository towards cooperation (i.e., the bound on the number of dependents)—was varied from 2 to 100 in our experiments.

5.2 Metrics

The key metric for our experiments is the 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 $|U(t)-S(t)| \leq tcr$ 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 ron is the mean fidelity of all repositories.

Rather than computing fidelity, our results plot a more meaningful metric, namely *loss in fidelity*. The loss in fidelity is simply $(100\% - \text{fidelity})$. Clearly, the lower this value, the better the overall performance of a dissemination algorithm.

In addition to fidelity, we also measure the number of updates (messages) sent by each dissemination technique. Clearly, the smaller the number of messages needed to maintain a certain fidelity, the lower the cost of the temporal coherency maintenance.

5.3 Experimental Results

5.3.1 Baseline Results

Our first experiment examines the efficacy of the three algorithms, namely SoPA, BePA and LeLA, to construct the ron. We used the unique coherency algorithm as the baseline data dissemination algorithm since, as shown in 5.3.4, it is the most efficient.

We assume a 400 node physical network and consider four different coherency scenarios: (i) $T = 100\%$ (all data items have stringent tcrs), (ii) $T = 80\%$ (80% of the items have stringent tcrs), (iii) $T = 20\%$, and (iv) $T = 0\%$ (all data items have loose coherency requirements). For each ron construction algorithm and these coherency tolerances, we vary the $\#cores$ from 2 to 100 and measure the efficacy of the resulting ron in providing good fidelity. Note that in the presence of the non-zero network delays, the structure of the ron has a significant impact on fidelity (since the data at a repository is out-of-sync until an update propagates through the ron and reaches the repository—the larger the end-to-end delay, the greater the loss in fidelity.

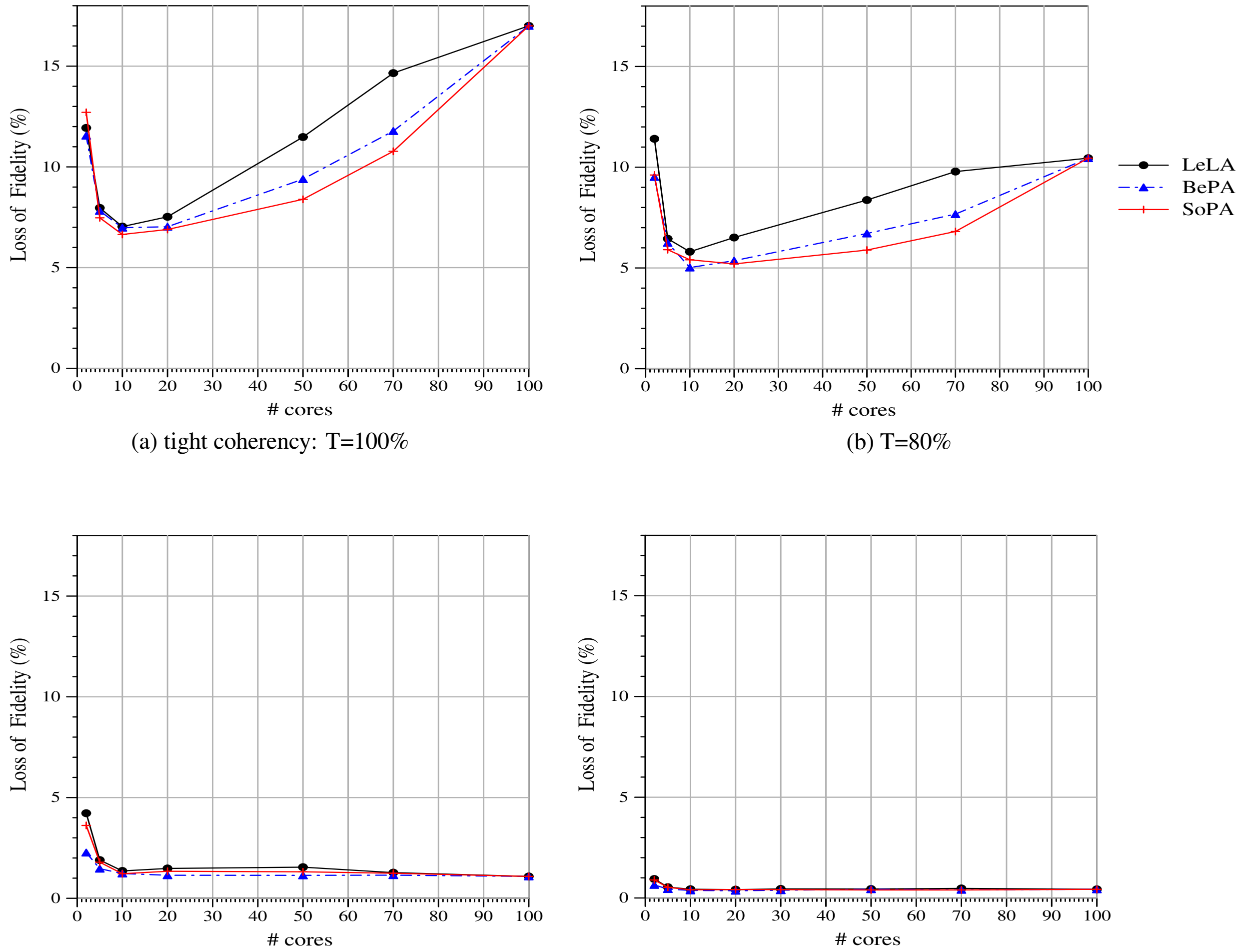
Figure 7 shows that all the algorithms exhibit a large loss of fidelity at low values of $\#cores$. The loss of fidelity occurs because the ron has a large diameter (i.e., a large number of hops between the source and the farthest repository), which increases the communication delays and decreases fidelity.

As the number of dependents of a repository (i.e., the $\#cores$) is increased, the loss in fidelity decreases to a minima and then starts increasing again. This is consistent with our expectations, since, as explained in Section 3, network delays dominate when there are a small number of dependents and processing delays at a repository dominate when there are a large number of dependents. The minima occurs when the sum of the two delays is minimized.

The point at which the minima occurs varies slightly from one algorithm to another and lies between 10 and 20 dependents per repository. Since all the algorithms aim to achieve high fidelity by reducing the total system overheads, i.e., link delays and computation delays, it is not surprising that their minima are close to each other. (It is worth pointing out that for these link and computational delays, the value of $actual\#cores$ is 12.)

Observe also that in the rising part of the curves, LeLA exhibits worse performance when compared to the other two algorithms. We attribute this to the fact that LeLA constrains its immediate search to repositories at a particular level and does not have the global perspective of SoPA or the global preference-based perspective of BePA.

The performance of the algorithms converge when the number of dependents per repository is increased beyond a certain value. This is because when the number of permitted dependents is large, the source serves most repositories



(a) tight coherency: T=100%

(b) T=80%

(c) T=20%

(d) Loose Coherency: T=0%

Figure 7: Effect of Temporal Coherency Requirements in Base Case

directly and the ron effectively reduces to a one-level tree with most repositories acting as a direct dependent of the source. We explore this behavior further in Section 5.3.2.

Note also that in Figures 7(a)-(d), as the fraction of data items with stringent coherency tolerances decreases, the gradient of the loss in fidelity also decreases. Furthermore, LeLA continues to exhibit worse performance than the other two algorithms, and there is no clear winner between BePA and SoPA since they swap their relative rankings as $\#cores$ is changed.

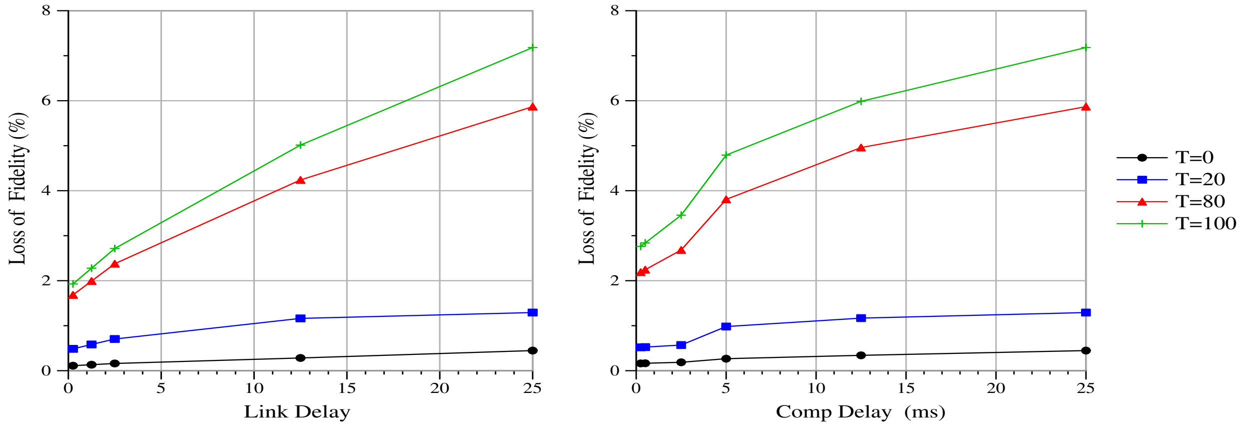
These results clearly show that, regardless of the ron construction algorithm, it is important for repositories to cooperate with one another to improve fidelity. Moreover, it is inappropriate to use a very large number of resources towards cooperation. Furthermore, with a proper choice of a bound on the number of dependents, the exact ron construction algorithm per se becomes less important. Hence, we address the issue of setting the "optimal" level of cooperation in the next section.

5.3.2 Effect of Cooperation on Fidelity

In this section, we thoroughly evaluate the effect of cooperation on fidelity. We begin by showing that if the server is entrusted with the task of disseminating updates directly to repositories, then a large loss of fidelity occurs, regardless of other system parameters. Thus, it is essential for the server to use the repositories to offload some of its dissemination overheads. We then examine the impact of three key parameters, namely the link delay, the computational delay and the factor C (see Eq. 8), on fidelity and demonstrate that when the number of dependents is adapted to link and computational delays, additional performance benefits can be harnessed

Performance in the Absence of Cooperation

In the previous section we have already shown that a scenario where the source directly disseminates updates to repositories (i.e., no cooperation between repositories) results in a large loss in fidelity. In this section we show that this result holds regardless of other system parameters. To demonstrate this, we vary the link delays and the computational delays and assume that the source directly services all repositories in the ron. We measure the fidelity offered by he source for different coherency tolerances ($T = 0\%$, $T = 20\%$, $T = 80\%$ and $T = 100\%$). Figures 8(a) and (b) depict our results. These figures show that there is a loss in fidelity regardless of the value of link and computational delays. Further, the loss in fidelity worsens with increasing link and computational delays, especially when coherency tolerances are stringent. This is because stringent coherency tolerances result in an increase in the number of messages being disseminated by the source, and since the server is entirely responsible for disseminating updates, fidelity suffers. The more stringent the coherency tolerance, the worse the loss in fidelity with increasing link and computational delays, and the greater the rate of decrease in fidelity. These results experimentally demonstrate that cooperation is essential to improve the fidelity of the dissemination process.



(a) Effect of Different Link Delays and T Values

(b) Effect of Different Computational Delays and T Values

Figure 8: Performance without Cooperation

Impact of Link and Computational Delays

Next, with controlled cooperation in effect, we study the impact of link and computational delays on fidelity. We vary both the computational delay and the link delay and examine the resulting loss in fidelity for the rons constructed using the three ron construction algorithms. Figures 9(a) and (b) depict our results. The figures show that increasing these delays causes updates to arrive at repositories later, which in turn worsens fidelity. However, adjusting the degree of cooperation *can help counter the effect of larger system delays.*

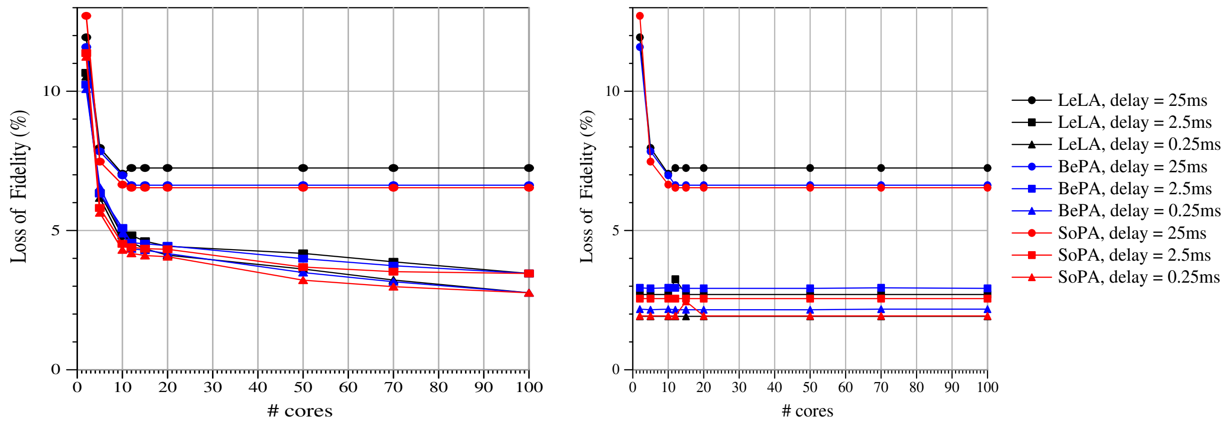
For example, with increasing computational delays, a smaller value of $actual\#cores$ is used (see Equation 8), and this reduces the load at a repository; on the other hand, a small value of computational delay results in a larger value of $actual\#cores$. This is the reason behind the curves for low computation delays in Figure 9(a) having a smaller fidelity loss with increasing $\#cores$ (these curves do not reach their minima within the $\#cores$ values plotted).

Similarly, with increasing link delays, a larger value of $actual\#cores$ is used and this will reduce the load on the network; on the other hand, a small value of link delay will result in a small value of $actual\#cores$. This is the reason behind the curves for low link delays in Figure 9(b) displaying a flat behavior starting from very low $\#cores$.

Thus, our results indicate that the degree of cooperation should be higher when the link delays are large and lower when the computational delays are large. This clearly demonstrates the benefits of adapting the choice of the $\#cores$ parameter to system overheads for providing high fidelity.

Sensitivity to the Parameter C

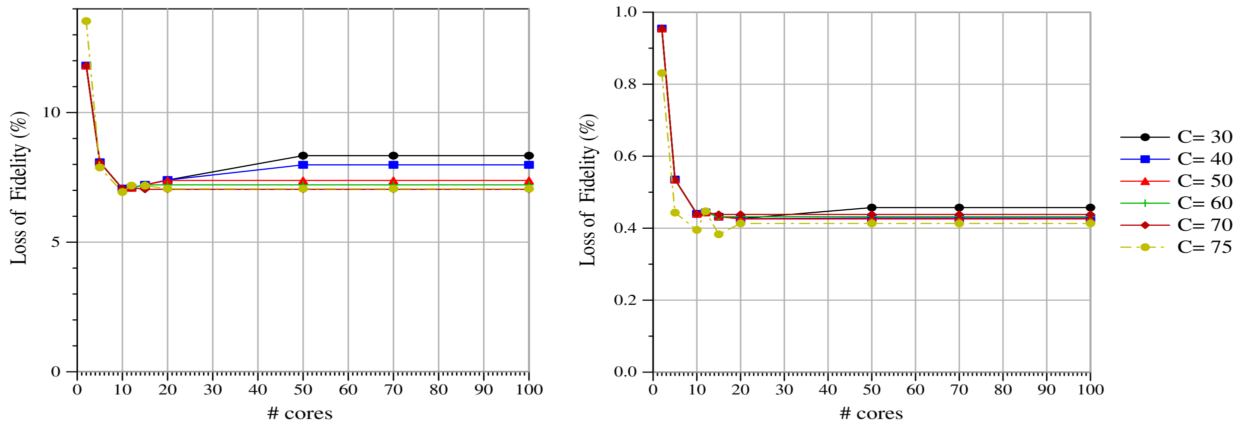
Having examined the impact of link and computational delays, we now examine the sensitivity of our results to the constant C used in Equation 8. Figure 10 shows the effect of varying the value of C on performance. The figure shows



(a) Effect of Computational Delays under Tight Coherency (b) Effect of Link Delays under Tight Coherency

Figure 9: Effect of Varying Link and Computation Delays

that the choice of C has only a small impact on fidelity since the same minima (i.e., optimal fidelity) can be achieved regardless of the actual value of C. In fact, the variation in fidelity loss is around 1% over a wide range of C values. This demonstrates that the parameter C has an insignificant impact on performance and that the degree of cooperation in Equation 8 should be chosen solely based on actual system overheads (i.e., link and computational delays).



(a) Tight Coherency (b) Loose Coherency

Figure 10: Effect of different C Values

5.3.3 Improvement in Fidelity When Coherency Requirements are taken into Account

A crucial assumption of our work is that only only updates of interest are disseminated by a repository to its dependent. In this section, we demonstrate that this assumption is, in fact, essential to achieving high fidelity. To demonstrate this aspect, 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 coherency tolerance and low computational overheads, causing all updates to be disseminated. We compare this system to one where the coherency tolerances are less stringent and the computation overheads are greater (less stringent tcrs result in filtering and selective forwarding of updates). Thus, any difference in performance between these systems is indicative of the fidelity improvement resulting from the filtering that occurs when repositories disseminate only data of interest to their repositories.

Figure 11 depicts our results. The figure shows that an approach that disseminate all updates, in fact, results in worse fidelity across a wide range of $\#cores$ values. This is because the approach disseminate more messages, which

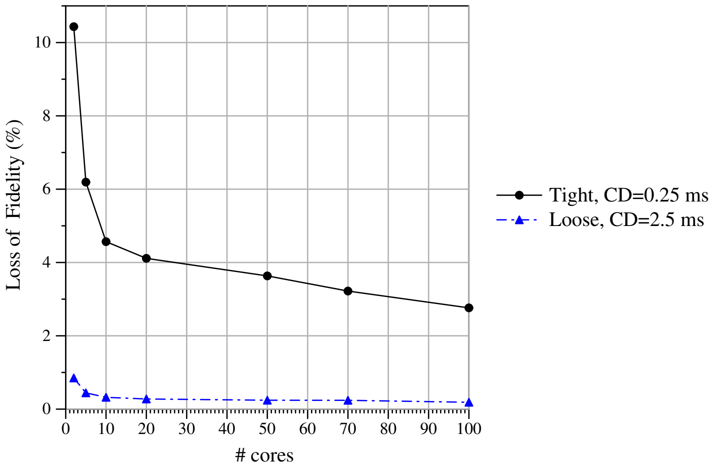
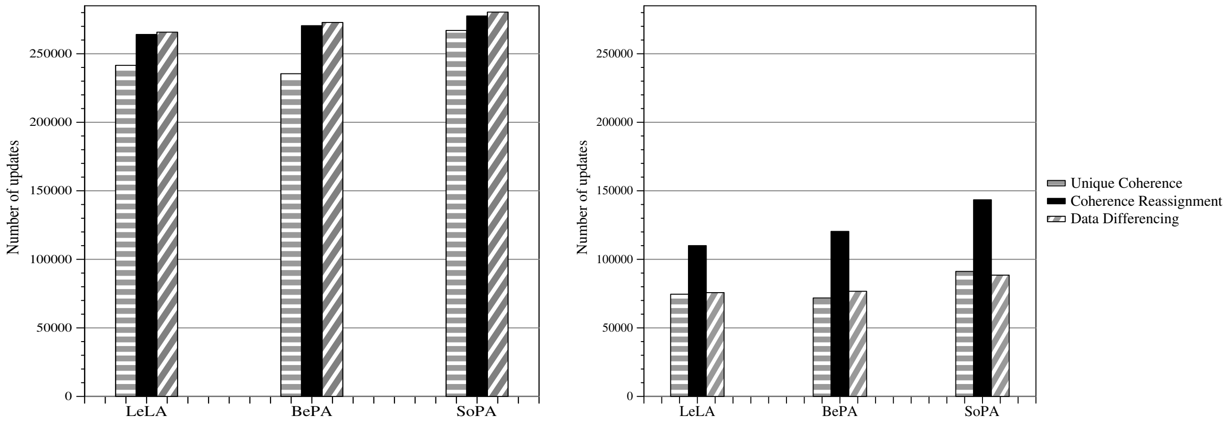


Figure 11: Loss in Fidelity if All updates are Disseminated

increases the network overheads as well as processing delays at repositories, causing a loss in fidelity. In contrast, intelligent filtering and selective dissemination of updates based on user's coherency tolerances can reduce overheads and improve fidelity.

5.3.4 Performance of Update Dissemination Algorithms

In this section, we compare the performance of our three dissemination algorithms, namely data differencing, coherence reassignment and unique coherence. Figure 12 compares the number of messages disseminated by the three algorithms for a fixed value of fidelity. The figure shows that the algorithms disseminate different number of messages to achieve the same fidelity. When the coherency requirements are tight, the difference in the number of updates disseminated by the three algorithms is less than 10% (since most updates need to be disseminated to meet these stringent requirements). The figure also shows the unique coherency algorithm outperforms the other two approaches.



(a) All repositories in the coherency range 0.01 - 0.1 (b) All repositories in the coherency range 0.1 - 0.9

Figure 12: Number of Messages sent by different Data Dissemination Algorithms

When coherency requirements are loose, the unique coherency algorithm continues to provide the best (or close to best) performance. However, the relative performance of the other two dissemination algorithms changes when compared to the tight coherency scenario.

Whereas unique coherence-based dissemination needs the least number of messages overall, coherence reassignment disseminates the most messages. This is because in the former case, the server transmits only those messages to a dependent which are of interest to that dependent or to one of its dependents. Coherence reassignment, on the other hand, is very conservative in the way it disseminates updates, and hence, ends up sending more updates than necessary. It is for these reasons that we choose unique coherence as our baseline temporal coherency algorithm.

5.3.5 Scalability of the algorithms

We have also studied the effect of increasing the number of repositories on fidelity. Whereas with unlimited cooperation, the diameter of the ron could grow to be very high with increasing number of nodes in the network, controlled cooperation limits this growth. For example, when the number of repositories grows from 100 (for the base case) to 250 (and with that the total number of nodes in the system grows from 400 to 1000 nodes), the increase in the loss in fidelity with controlled cooperation was observed to be between 1% and 5%. This is indicative of the scalability of our set of solutions.

5.4 Summary of Experimental Results

To achieve high fidelity during the transmission of dynamically changing data to repositories, it is important (a) for the server to shed some its data dissemination load to other repositories and (b) for the repositories to cooperate with each other during data dissemination. Specifically, it is essential to form an overlay network over the physical topology connecting the repositories so as to minimize the system-wide computational and communication delays incurred during the dissemination of updates from servers to repositories. This overlay network should take network link delays, delays for pushing individual updates, the coherency required, the number of resources a repository is willing to provide towards cooperation, etc. Our specific results indicate that

- Each repository should disseminate only updates of interest to its dependents.
- Cooperation is essential to achieve low fidelity.
- Cooperation beyond a certain point leads to loss of fidelity. This is because if a repository agrees to disseminate data to too many dependents, queue-ing and dissemination delays at that repository can reduce the fidelity achieved.
- For low link delays and low computational delays, most of the algorithms behave the same at all levels of cooperation. But when these delays are not negligible, the degree of cooperation should be chosen taking link and computational delays into account because outside the optimal value, the algorithms could lead to increased loss of fidelity.

6 Related Work

Research on replication in traditional databases [26] has focused achieving transactional consistency among replicas (for example, see [19]). However, the notion of temporal coherency defined in this paper requires a different architecture and algorithms than those used in replicated databases.

Efforts that are more relevant to our work have dealt with maintaining consistency between sources and cached copies. [3] is one of the earliest papers on consistency maintenance between a data source and (cached) copies of the data. The paper discusses techniques whereby data sources propagate (push) updates to clients based on expiration times associated with data. Several other research efforts have also investigated push-based techniques. These include broadcast disks [2], continuous media streaming [4], publish/subscribe applications [25, 5], web-based push caching [18], and speculative dissemination [6]. Note, however, that these efforts are not explicitly targeted at disseminating continuously changing data with associated coherency tolerances.

More recently, the issue of consistency maintenance has been investigated in the context of caching in the World Wide Web. These efforts fall into two categories: (i) weak consistency mechanisms, where cached data can be out-of-sync with servers, and (ii) strong consistency mechanisms, where cached data is always up-to-date with servers. Examples of the former include client polling [13], where a client periodically polls the server for updates to the data, and adaptive time-to-live (TTL) values [32], where objects are refreshed based on the rate of change of the data. Examples of the latter include server invalidation [23], where the server sends sends invalidation messages to all clients when an object is modified. These efforts either assume a single repository, or that cached data is modified on slow time scales (e.g., tens of minutes or hours). Hence, they are less effective at maintaining consistency of rapidly changing data in multiple (cooperating) repositories.

The issue of cooperative caching has also been investigated in the context of the web, albeit for predominantly static web content. These efforts include the design of hierarchical caching architectures [1, 7] and the issue of meta-data maintenance in such hierarchies [15, 33]. Other efforts include: (i) the use of overlapping groups of cooperating caches [38], (ii) the use of cooperation to handle caching and transcoding of data for mobile clients [22], and (iii) the use of leases to maintain consistency in caching hierarchies [35, 36].

Efforts that focus on *dynamic* web content include [21] where push-based invalidation and dependence graphs are employed to determine where to push invalidates and when. The difference between this approach and ours is that repositories don't cooperate with one another to maintain coherency. Other efforts in this area include adaptive replica placement in content distribution networks [28, 17], distributed servers [12], distributed cache placement [34], and technique to achieve scalability and availability by adjusting the coherency requirements of data items [20]. In contrast to these efforts, our work has focused on the construction of a cooperating repository overlay network so as to provide high fidelity to time-varying data.

Recent efforts on application-level multicast are also related to our work. Observe that each repository effectively uses application-level multicast to push updates to its dependents. Work on application-level multicast has focused on the design of multicast routing and tree construction algorithms at the application layer [27, 16, 9]. These efforts are generic, in the sense that any application requiring the use of application-level multicast can use them. In contrast, our tree construction algorithm and multicast-based dissemination are specifically optimized for the problem at hand, namely maintaining coherency of time varying data. For instance, our tree construction algorithms take into account application-specific characteristics such as coherency requirements of data items and the processing delays at a repository, and hence, can generate an overlay network that is more effective than those generated by more generic algorithms.

Mechanisms for disseminating fast changing documents are proposed in [31, 30, 29]. The technique, termed as Continuous Multicast Push, requires receivers to tune to a multicast group to receive updates in a reliable manner and then leave the group once updates have been received. The difference though is that receivers 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 (which reduces the burden on the infrastructure).

Multicast-based invalidations and volume leases are incorporated in a caching hierarchy in [37]. While sharing our goals by studying the tradeoffs between communication between nodes, update rate at the server and staleness, this work does not examine the generation of cache hierarchies to maximize fidelity and does not employ any specific coherency semantics to measure staleness.

Finally, our work can be seen as providing support for executing continuous queries over dynamically changing data [24, 8]. Continuous queries in the Conquer system [24] are tailored for heterogeneous data, rather than for real time data, and uses a disk-based database as its backend. NiagraCQ [8] focuses on efficient evaluation of queries as opposed to temporally coherent data dissemination to repositories (which in turn can execute the continuous queries). This approach leads to better scalability.

In summary, none of the research efforts have focused on the infrastructure necessary for the temporally coherent dissemination of time-varying data using cooperating repositories. Specifically, our approach (a) defines a coherence mechanism for the dissemination architecture, (b) creates a definite flow structure for time-varying data to provide high fidelity at a low cost, and (c) addresses the problem of efficient placement of objects in the repositories by taking into account the loads and processing delays at various repositories.

7 Conclusions

In this paper, we examined the design of an overlay network of repositories that cooperate with one another to maintain temporal coherency of time-varying data. 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 repositories, which reduces the system-wide network overhead as well as the load on repositories. These in turn improve the fidelity of data stored at repositories.

- Design of mechanisms for maintaining temporal coherency of data within an overlay network of repositories.

Our mechanisms were designed to take into account network delays, computational overheads, and the system load. We also studied their relative performance and showed that cooperation among repositories can be used to improve fidelity substantially with lower overheads, but beyond a certain point, altruism can be detrimental to performance.

Traditionally, cooperative architectures have been employed to place replicas of objects closer to the clients so as to minimize access latency and the server load. In our approach, we employ cooperation to provide a data flow to clients and to alleviate the source from directly handling all the clients Thus, cooperation is used to route and distribute state information, rather than to replicate and cache data items. Even though this is akin to the goals of application-layer multicast, we are also concerned with achieving temporal coherency, and hence, take link and computational delays into account to improve fidelity.

Finally, whereas our approach uses push-based dissemination, other dissemination mechanisms such as pull [32], adaptive combinations of push and pull [11], as well as leases [14] 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] Squid internet object cache users guide. available online at http://squid.nlanr.net.

[2] 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.

[3] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Systems*, September 1990.

[4] E. Amir, S. McCanne, and R. Katz. An active service framework and its application real-time multimedia transcoding. In *Proceedings of the ACM SIGCOM Conference*, September 1998.

[5] 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.

[6] 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.

[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] Y. Chu, S. Rao, and H. Zhang. A case for endsystem multicast. *Internet End-to-End Research Meeting*, June 1999.

[10] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. Mc-Graw Hill, 1990.

[11] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramaritham, and P. Shenoy. Adaptive push pull of dynamic web data: Better resiliency, scalability and coherency. In *Procs of WWW 10*, May 2001.

[12] D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari. A scalable and highly available server. In *Proceedings of the IEEE Computer Conference (COMPCON)*, March 1996.

[13] A. Dingle and T. Partl. Web cache coherence. In *Proc Fifth Intnl WWW Conference*, May 1996.

[14] V. Duvvuri, P. Shenoy, and R. Tiwari. Adaptive leases: A strong consistency for the world wide web. In *Proceedings of IEEE INFOCOM 2000*, March 2000.

[15] L. Fan, P. Cao, J. Almeida, and A. broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the ACM SIGCOMM*, 1998.

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

[17] S. Gadde, J. Chase, and M. Rabinovich. Web caching and content distribution: A view from the interior. In *In Proc. of the Fifth Int. Web Caching and Content Delivery Workshop, May*, 2000.

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

[19] JoAnne Holliday, Divyakant Agrawal, and Amr El Abbadi. Database replication using epidemic communication. In *6th International Euro-Par Conference*, September 2000.

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

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

[22] R. Kavasseri, T. Keating, M. Wittman, Anupam Joshi, and S. Weerawarana. Web intelligent query - disconnected web browsing with collaborative techniques. In *Proceedings First IFCIS Conf. on Cooperative Information Systems*, 1997.

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

[24] 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.

[25] 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.

[26] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd edition*. Prentice-Hall, 1999.

[27] Dimitrios Pendarakis, Sherla Shi, Dinesh Verma, and Marcel Waldvogel. Almi: An application level multicast infrastructure. *In Proceedings of 3rd USENIX Symposium on Internet Technologies and Systems (USITS'01), San Francisco*, March 2001.

[28] Michael Rabinovich and Amit Aggarwal. Radar: A scalable architecture for a global web hosting service. *WWW8 / Computer Networks*, 31(11-16):1545–1561, 1999.

[29] P. Rodriguez and E. Biersack. Continuous multicast push of web documents over the internet. *IEEE Network Magazine, vol. 12, 2, pp. 18–31, Mar-Apr*, 1998.

[30] P. Rodriguez, E. Biersack, and K. Ross. Automated delivery of web documents through a caching infrastructure. *Technical Report, EURECOM, June*, 1999.

[31] Pablo Rodriguez, Keith W. Ross, and Ernst W. Biersack. Improving the WWW: caching or multicast? *Computer Networks and ISDN Systems*, 1998.

[32] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual warehouses. In *Proceedings of of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[33] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report CS98-04, Department of Computer Science, UT Austin Austin, Texas, USA, 1998.

[34] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth constrained placement in a wan. *In ACM Principles of Distributed Computing*, August 2001.

[35] 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.

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

[37] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *SIGCOMM*, pages 163–174, 1999.

[38] Lixia Zhang, Scott Michael, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson. Adaptive web caching: Towards a new global caching architecture. *3rd International Caching Workshop*, June 1998.

Appendix A: Derivation of Equation 6

In Section 2, we discussed the problem of assigning coherencies to the proxies in a proxy hierarchy such that no proxy misses any updates that violate its coherence requirement.

Here, we show that Equation 5:

$$\mid u^s_{i+1} - u^p_j \mid - \mid u^s_{i+1} - u^q_k \mid \quad < \quad c^p - c^q \tag{13}$$

reduces to Equation 6:

$$c^q \quad - \quad \mid u^p_j - u^q_k \mid \quad < \quad c^p \tag{14}$$

given the conditions of Equations 3 and 4:

$$\mid u^s_{i+1} - u^p_j \mid \quad < \quad c^p \tag{15}$$

$$\mid u^s_{i+1} - u^q_k \mid \quad \geq \quad c^q \tag{16}$$

Also Equation 17 holds at all times:

$$\mid u^p_j - u^q_k \mid \quad < \quad c^q \tag{17}$$

Let us consider the following cases for equation 13

Case 1:

Let, $u^s_{i+1} > u^p_j$ and $u^s_{i+1} > u^q_k$.

In this case it is trivial to see to see that equation 13 reduces to 14

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

Case 2:

Let, $u^s_{i+1} > u^p_j$ and $u^s_{i+1} < u^q_k$.

the above condition can be combined into

$$u^q_k \quad > \quad u^s_{i+1} \quad > \quad u^p_j \tag{18}$$

Now from equation 18 and equation 16 we get

$$u^q_k - u^p_j \quad > c^q \tag{19}$$

which is in contradiction with equation 17. Hence this case is not possible at all. We can similarly argue for the last case where $u^s_{i+1} < u^p_j$ and $u^s_{i+1} > u^q_k$.

Hence, we can see that Equation 13 reduces to 14