Efficient Execution of Continuous Threshold Queries over Dynamic Web Data

Manish Bhide, Hrishikesh Gupta, Krithi Ramamritham, and **Prashant Shenoy**†

Laboratory for Intelligent Internet Research
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076, India.
{manishb,hkgupta,krithi}@cse.iitb.ac.in

†Department of Computer Science
University of Massachusetts,
Amherst, MA 01003 USA.
shenoy@cs.umass.edu

November 13, 2001

Abstract

On-line decision making often involves processing significant amount of time-varying data. Examples of time-varying data available on the Web include financial information such as stock prices and currency exchange rates, real-time traffic, weather information and data from process control applications. In such environments, typically a decision is made whenever some function of the current value of a set of data items satisfies a threshold criterion. For example, when the traffic entering a highway exceeds a pre-specified limit some flow control measure is initiated; when the value of a stock portfolio goes below a comfort level, an investor might decide to rethink his portfolio management strategy. In this paper we develop data dissemination techniques for the Web where such queries access data from multiple sources. Key challenges in supporting such *Continuous Threshold Queries* lie in meeting users' consistency requirements while minimizing network and server overheads, without the loss of fidelity in the responses provided to users. We also show the superior performance of our techniques when compared to alternatives based on periodic independent polling of the sources.

1 Introduction

1.1 Motivation

An increasing fraction of data on the web is time-varying (i.e., varies continuously with time). Examples of time-varying data includes financial information such as stock prices and currency exchange rates, real-time traffic, weather information and data from process control applications. Time-varying web data is frequently used for online decision making, and in many scenarios, such decision making involves *multiple* time-varying data items, possibly from multiple independent sources. Examples include a user tracking a portfolio of stocks and making decisions based on the net value of the portfolio. Observe that computing the overall value of the portfolio at any instant requires up-to-date values of each stock in the portfolio. In some scenarios, users might be holding these stocks in different (brokerage) accounts and might be using a third-party aggregator, such as yodlee.com, to provide them a unified view of all their investments. The third-party aggregator assembles a unified view of the portfolio by periodically gathering and refreshing information from each individual account (source), all of which may be independent of one another. If the user is interested in certain events (such as a notification every time the value of the portfolio increases or decreases by \$1000), then the aggregator needs to periodically and intelligently refresh the value of each stock price to determine when these user-specified thresholds are reached. More formally, the aggregator runs a *continuous query* over the portfolio by intelligently refreshing the components of the portfolio

from their sources in order to determine when user-specified thresholds are exceeded. We refer to such queries as *Continuous Threshold Queries (CTQ)*. Other examples of CTQs include: a decrease in the value of a stock portfolio below a certain level might trigger a review of the portfolio investment strategy, or an increase in the traffic at an intersection of a busy thoroughfare beyond a pre-specified limit might cause flow control measures to be initiated.

In this paper, we consider techniques for efficiently executing continuous threshold queries at a proxy. Much of the prior work has assumed that it is difficult to cache time-varying data at a proxy, and consequently, requests for these data items as well as queries on these data items are handled directly by the server (data source). Such an approach has two important limitations. First, since continuous threshold queries are compute-intensive (due to the need to continuously evaluate the query condition), the server could become a bottleneck, and thereby, limit the scalability of the system. Second, a pure server-based approach makes CTQs on data items from multiple independent sources impossible (for example, the server-based approach is unfeasible in the case where a user specifies a query on a stocks held in multiple independent accounts). Executing queries at a proxy eliminates both restrictions—the approach is scalable, since computations are offloaded from the server to proxies, and queries on data items from multiple sources become feasible. In addition, a proxy-based approach improves user response times [21, 22]. However, such an approach raises new research challenges. The key challenge is to ensure that the results of the query are no different from the case where the query is handled by the server (i.e., correctness of the results should be ensured). To do so, the proxy should ensure that cached values of time-varying data items are *temporally coherent* with the source. In this paper, we address these challenges by presenting (i) a suite of data dissemination algorithms to efficiently disseminate time-varying data items from servers to proxies (observe that a naive approach that disseminates every update to a frequently changing data item can have a prohibitive overhead), and (ii) techniques to maintain coherency of cached, time-varying data items with the server. Together, these techniques enable an efficient implementation of CTQs at a proxy.

In the rest of this section, we provide a precise definition of the notion of continuous threshold queries and then outline the research contributions of this paper.

1.2 Continuous Threshold Queries: An Introduction

A CTQ $Q(\mathcal{D},\mathcal{N},\mathcal{T})$ operates on data items $d_1, \ldots, d_N \in \mathcal{D}$ and informs the invoker whenever the following inequality holds.

$$\sum_{i=1}^{i=n} d_i(t) \times n_i \geq \mathcal{T} \tag{1}$$

where $\mathcal{T}$ is a threshold, $n_1, n_2, \ldots, n_N \in \mathcal{N}$ are the weights attached to the data items. The left hand side of the inequality is referred to as the *value of the CTQ*. So using such a CTQ a user can be informed whenever a CTQ value exceeds the specified threshold.

Thus, in our example of portfolio tracking, the user is notified whenever the value of the stocks at the source exceeds a specified threshold. That is, whenever

$$\sum_{i \in portfolio} S_i(t) \cdot n_i \geq threshold \tag{2}$$

where $S_i(t)$ denotes the current price of stock i at the source and n_i denotes the number of stocks of i held in the portfolio. In the event that this CTQ is executed at a proxy rather than the server (source), then the following inequality should hold at the proxy whenever the above inequality holds at the server:

$$\sum_{i \in portfolio} P_i(t) \cdot n_i \geq threshold \tag{3}$$

where $P_i(t)$ denotes the (cached) value of stock i at the proxy at time t. Hence the proxy needs to dynamically track changes in the stock price to successfully execute the CTQ.

1.2.1 Fidelity of CTQs

We use a metric refereed to as *fidelity* to measure the accuracy of the CTQ executed by a proxy with that at the server. Two kinds of inaccuracies can result when a CTQ is executed by a proxy: *(i) false positives*, where the proxy detects that the threshold has been exceeded when, if fact, it hasn't, and *(ii) false negatives*, where the threshold is exceeded at the server but the proxy fails to detect it. Between the two, it is more challenging and more important to eliminate (or at least minimize) false negatives—false positives can always be re-verified by simply refreshing the values of all cached data items from their sources, but no such solution is available for false negatives.

To quantify the misperceptions, we define *fidelity* of the portfolio to be the total length of time for which the CTQ state at the client is correct, normalized by the total length of time for which the CTQ is executed In addition to fidelity, another measure of evaluating the mechanism is to measure the number of false positives and false negatives that have occurred.

1.3 Coherency of Data Items

Notice that if the CTQ at the server (i.e., Eq 1) is not satisfied at a particular instant t, then the exact values of data items cached at the proxy, $P_i(t)$, *need not correspond to* $S_i(t)$*, the values at the server* (it is sufficient for the proxy to know that the CTQ is not satisfied). In other words, the proxy needs to be aware of only those updates to the data items that cause the CTQ value to exceed the threshold at the server. This key observation allows us to reduce the number of updates to data items that need to be propagated from the server to the proxy—by only propagating "interesting" updates, i.e., those likely to cause the CTQ's value to exceed the threshold, a proxy can efficiently execute CTQs with a minimal loss of accuracy. Further, the coherency requirements of individual data items can be derived from the specification of the $\mathcal{T}$ associated with the CTQ, where the derived data coherency requirements are sufficient to satisfy the CTQ's threshold specifications.

1.4 Contributions of this Paper

In this paper, we introduce a new class of continual queries, referred to as continual threshold queries, and present efficient techniques to implement CTQs at a proxy. To achieve this objective, we address the following important questions.

1. How should one derive the coherency requirement of each of the cached data items used by the CTQ ?

   The coherency requirement associated with a data item depends on (a) the overall contribution of the data item to the CTQ's result and (b) the threshold value associated with the CTQ. In this paper we develop techniques to determine the data coherency requirements to ensure correct execution of the CTQs.

2. How should the (derived) coherency requirement associated with each item be ensured?

   To maintain coherency of the cached data so that, each cached item must be periodically refreshed with the copy at the server. In this paper we demonstrate that, in the case of CTQs, using standard techniques, such as pull and push, for individually retrieving each of the data items is not efficient - *additional mechanisms must be employed which exploit the fact that the data items that are needed to execute a CTQ form a semantic unit.*

Note that a user is interested only in the fidelity of the CTQ and not the exact values of the d_i's. Hence as long as a proxy is correctly informed about the value of a CTQ, even if a d_i is changing very rapidly there is no need to follow these changes frequently. This is a key observation that is exploited by our novel techniques.

The *Pull-Based CTQ Execution Approach ($PullCEA$)* outlined in Section 2 makes use of this fact to reduce the network overhead without loss of fidelity. Here, a proxy dynamically changes the coherency requirement of a data item so that, indirectly, the TTR calculated by the Adaptive TTR algorithm is adjusted based on whether or not the value of the CTQ is close to the threshold. That is, one possible approach to the two problems mentioned earlier in

this section is to use the proxy-based $PullCEA$ to dynamically choose the coherency of individual data items and use the *Adaptive TTR Algorithm* to maintain the coherency requirements.

An alternative is to use a *Push-Based CTQ Execution Approach ($PushCEA$)* with push-based sources, outlined in Section 3. Here a proxy calculates the coherency requirement of data items and communicates them to the data sources. In contrast with pull-based techniques, these offer better fidelity for individual data items and hence for the CTQs.

Unfortunately, a push based approach requires that a source should maintain the coherency requirements of the data items that it serves. This consumes state space at the server and is also prone to scalability and resiliency problems. An intelligent combination of these two is hence called for and is the subject of Section 4. We develop a high performance *Hybrid CTQ Execution Approach (HyCEA)* that dynamically categorizes CTQ's data items into those needing pull vs. those needing push. These are aimed at reducing the state space requirements and resiliency and scalability problems of a push based approach while retaining its fidelity maintenance properties.

In Sections 2 through 4 we describe the three approaches and experimentally evaluate their performance using CTQs defined over real-world traces of dynamically changing data (specifically, stock prices). Our evaluations show that by dynamically tracking the changes to data item values and by judicious combination of push and pull approaches we can obtain the best of all possibilities and achieve high fidelity and scalability at low network overheads and source loads.

We end the paper with a discussion of related work in Section 5 and a summary of this paper in Section 6.

2 The Pull-Based CTQ Execution Approach ($PullCEA$)

$PullCEA$ is entirely pull-based and does not need any special support at servers (and hence, is compatible with any HTTP server). The algorithm dynamically computes the coherency requirements of each data item based on its "contribution" to the overall value of the CTQ—data items that contribute a larger fraction of the overall value are given a tighter coherency requirement, denoted henceforth as $\mathcal{C}$, which enables the proxy to track that data item with greater accuracy. The algorithm uses $\mathcal{C}$s to determine how frequently to refresh (pull) the new value of each data item—data items with tighter (smaller) coherency requirements are refreshed more frequently.

2.1 Initial Allocation of Coherency Requirements to Data Items

After a new CTQ is specified to the proxy, our algorithm first determines the coherency requirement for CTQ's data items. To do so, our algorithm determines the potential amount by which each data item needs to change in order to contribute to an overall change of $\mathcal{T}$ in the value of the CTQ. Thus, given the threshold $\mathcal{T}$, our algorithm apportions a part c_i to each data item such that

$$c_1 \times n_1 + c_2 \times n_2 + \ldots + c_N \times n_N = \mathcal{T} \tag{4}$$

The parameter c_i is the $\mathcal{C}$ of data item i. Intuitively, if each data item changes by amount c_i, CTQ's value changes by $\mathcal{T}$. The challenge then is to determine an appropriate c_i for each data item such that Equation 4 is satisfied. Our algorithm determines c_i based on the overall contribution of the data items to the value of the CTQ—data items that have a larger contribution are given a smaller coherency requirement, implying that the proxy tracks those data items with greater accuracy. Assuming a weight n_i for data item i and p_i as its value at the proxy, the weighted value of the item is

$$v_i = n_i \times p_i \tag{5}$$

In our stock portfolio example, v_i denotes the total value of the ith stock, assuming that n_i stocks, each with a value p_i are held.

In such a scenario, the coherency requirement c_i is computed as

$$c_i = \frac{\sum_{j=1}^{n} v_j - v_i}{\sum_{j=1}^{n} v_j} \times \frac{\mathcal{T}}{n_i \times (N-1)} \quad (6)$$

The larger the contribution of data item i (i.e., v_i), the smaller the quantity $\frac{\sum_{j=1}^{n} v_j - v_i}{\sum_{j=1}^{n} v_j}$, and the smaller the resulting tolerance c_i. The second factor $\frac{\mathcal{T}}{n_i \times (N-1)}$ ensures that the computed c_is satisfy Equation 4.

To illustrate this process, consider a stock portfolio CTQ with $\mathcal{T} = 90$.

Company A Data	*Company B Data*
Number of stocks held $n1$ = 100	Number of stocks held $n2$ = 200
Price of each stock $p1$ = \$10	Price of each stock $p2$ = \$20

Using the above formula the coherency requirements are computed as follows:

$$c_1 = \frac{20 \times 200}{10 \times 100 + 20 \times 200} \times \frac{90}{100} = 0.72$$
$$c_2 = \frac{10 \times 100}{10 \times 100 + 20 \times 200} \times \frac{90}{200} = 0.09$$

Since the portfolio comprises of a larger number of stocks of company B, each of which is also priced higher, the coherency requirement of stock B is more stringent than that of stock A.

2.2 Dynamic Adjustment of Coherency Requirements

After computing the initial coherency requirement c_i for each data item, our algorithm adapts the coherency requirement to the dynamics of the data. Thus, the coherency requirement is decreased or increased depending on whether the updates to the data item takes the CTQ towards or away from its threshold (i.e., $\mathcal{T}$). The algorithm consider four possible scenarios.

- *Scenario 1: The value of a data item increases by more than its coherency requirement c_i.* This triggers a recomputation of the coherency requirements of all data items as outlined in Section 2.1 using a new threshold $\mathcal{T} - c_i \cdot n_i$ — since the query is closer to its threshold $\mathcal{T}$ by an amount $c_i \cdot n_i$, all coherency requirements are made more stringent to enable more accurate tracking. This also reduces the chances of false negatives.

- *Scenario 2: Changes to a data item take the query away from its threshold $\mathcal{T}$.* Such changes are less interesting to a proxy, since it will take longer for the threshold $\mathcal{T}$ to be triggered. Specifically, the accuracy with which the object is tracked is reduced by making the coherency requirement less stringent. Thus, c_i is assigned $c_i \times \gamma$, where $\gamma > 1$.

- *Scenario 3: The data item is changing very slowly.* Since there are relatively few changes to the data item, the proxy can relax the coherency requirement for the data item. Specifically, if the change in the value of the data item has not exceeded its coherency requirement even once within a timeout interval δ, then its coherency requirement is increased as $c_i = c_i \times \gamma$, where $\gamma > 1$.

- *Scenario 4: The data item sees a sudden change after a long period of no changes.* Since the coherency requirement is gradually relaxed when an object changes slowly, a subsequent sudden change causes the proxy to sharply reduce the coherency requirement. Thus, the requirement is decreased as $c_i = c_i \times \beta$, where $0 \leq \beta \leq 1$.

To ensure that the coherency requirement does not take very large values (which can cause a proxy to miss changes of interest), our algorithm imposes an upper bound on the value of c_i. Hence, after each of the above scenarios, the algorithm ensures that $c_i = \min(c_i, c_{max})$, where c_{max} is a pre-specified upper bound.

After computing the coherency requirements, the proxy uses these requirements to determine how frequently to refresh (poll) each data item from the server. The more stringent the requirement, the more frequent the refresh. This is achieved using the following algorithm proposed in an earlier work [19].

2.3 The Adaptive TTR Algorithm

To maintain coherency of individual data items, a proxy computes a *Time To Refresh (TTR)* for each data item. The TTR denotes the next time that the proxy should refresh the data item from the server. The algorithm ensures that the difference in the value of a data item at the server and a proxy is bound by the coherency requirement c_i. That is, $\forall t,\ |S_i(t) - P_i(t)| < c_i$. Our algorithm [19] provides these guarantees by polling the server every time the value of the data changes by c_i. This is achieved by computing the rate at which the data value changed in the recent past and extrapolating from this rate to determine how long it would take for the value to change by c_i. Thus, the TTR is estimated as

$$TTR = c_i/r \tag{7}$$

where r denotes the rate of change of the data value and is computed as $r = \frac{|P_i(t_{curr}) - P_i(t_{prev})|}{t_{curr} - t_{prev}}$; t_{curr} and t_{prev} denote the times of the two most recent polls and $P_i(t)$ denotes the data values obtained from those polls. This TTR estimate can be improved by accounting for changes that occurred prior to the immediate past; this is achieved using an exponential smoothing function to refine the TTR value. That is, $TTR = w \cdot TTR + (1 - w) \cdot TTR_{prev}$, where w determines the weight accorded to the current and past TTR estimates. This TTR value is then constrained using static upper and lower bounds and weighed against the smallest observed value of TTR thus far. Thus,

$$\begin{aligned} TTR = max(TTR_{min}, min(TTR_{max}, f \cdot TTR + \\ (1 - f) \cdot TTR_{observed-min})) \end{aligned} \tag{8}$$

where f is a tunable parameter, $0 \leq f \leq 1$.

Since this algorithm provides good fidelity for individual items and incurs a low network overhead [19], we use it in our $PullCEA$ algorithm.

In the next two sections we present the results of experiments conducted to evaluate the efficiency of our approach. We first present our experimental methodology and then our experimental results.

2.4 Experimental Methodology

2.4.1 Simulation environment

Our experiments are conducted with stock portfolio tracking as examples of CTQs. All algorithms were evaluated using a prototype server/proxy that employed trace replay. Our experiments assume that the proxy has an infinitely large cache to store data and that the network latency in polling and fetching data items from the server is fixed (this is because we are primarily interested in comparing network overheads based on the number of messages transmitted). We assume that the value of C_{max} for $PullCEA$ is specified by the user. The proxy can keep track of the maximum change in the data value seen so far and that can also be used for C_{max}.

2.4.2 Traces used

The performance of the algorithm was evaluated using real-world traces. The presented results are based on data value traces (i.e., history of data values) of a few companies (Table 1) obtained from http://finance.yahoo.com. The traces were collected at a rate of 2-3 stock traces per second and can be considered to be real time traces. All the experiments were done on the local intranet. To get some consistency with the results, synchronization was done between access to different data sources. The number of stocks per portfolio was varied from 3 to 8 stocks.

2.4.3 Metrics

The algorithm was evaluated using the following metrics (i) number of polls (normalized by the length of the trace) (ii) Fidelity of the portfolio. Fidelity can be measured based on the total time duration for which the proxy was

Table 1: Traces used for the Experiment

Company	#Items	Max Value	Min Value
Microsoft	7078	65.875	64.625
Veritas	10000	135.75	131.50
DELL	6852	43.75	42.87
CBUK	10000	8.625	8.25
INTC	10000	134.50	132.50
Cisco	10000	65.0	63.5
UTSI	4136	22.25	21.0

oblivious of the correct state of the portfolio.

$$f = 1 - \frac{Total\ out\ of\ sync\ time}{Total\ trace\ duration}$$

where the *Total out of sync time* is the time duration for which the proxy was oblivious of the right state of the portfolio i.e., the time duration for which there were *false negatives* or *false positives*.

2.5 Experimental Results of $PullCEA$

The $PullCEA$ was evaluated using the traces mentioned in table 1. To do so (unless otherwise stated) the algorithm was configured using the parameters stated in table 2. The results were also compared with an approach in which the $\mathcal{C}$s were calculated only initially and were kept unchanged irrespective of the relative changes in the values of the data items. For evaluation of the algorithm the median of the portfolio was found offline and the threshold was set to the median of the portfolio. The portfolio value crosses the median value maximum number of times and evaluating the portfolio at this threshold helps us understand the behaviour of the algorithms in a better way.

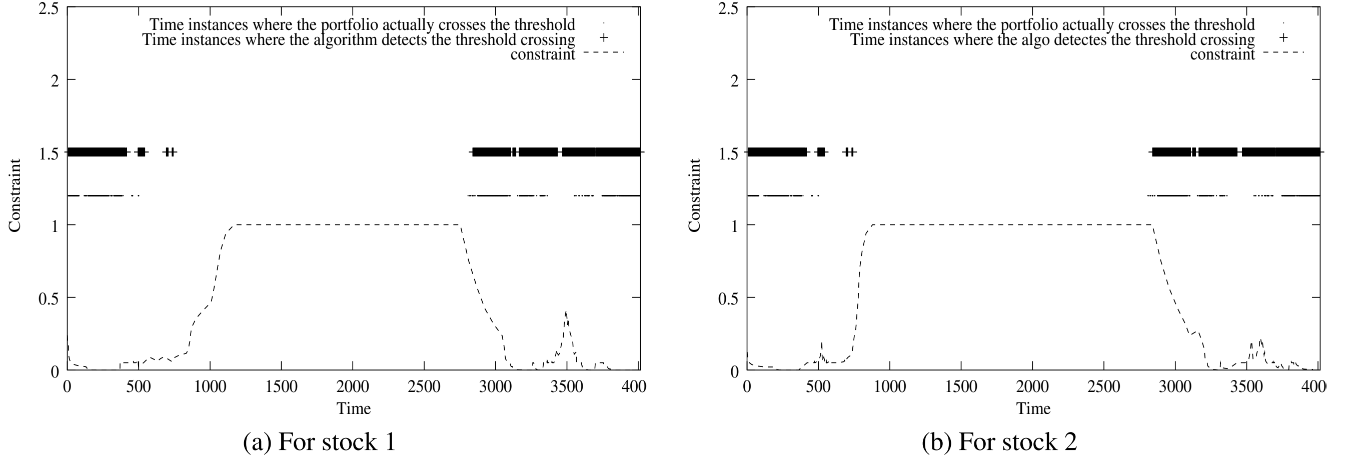
We begin considering a simple portfolio comprising stocks of two companies, with 200 stocks of the first and 300 stocks of the second, with the total value being :

$$200 \times price\ of\ stock1 + 300 \times price\ of\ stock2$$

Figure 1 shows shows the variation of the $\mathcal{C}$ with time for the two stocks. Also shown in the two figures are the points in time when the portfolio (a) actually crosses the threshold (dots) and (b) the points in time when the portfolio crossed the threshold at the proxy as per the $PullCEA$ algorithm (For these two curves there is no significance to Y-axis).

Symbol	Meaning	Value
δ	The time in seconds that governs the change in $\mathcal{C}$ value	60 seconds
γ	Factor by which the $\mathcal{C}$ is increased if (a) Changes to a data item take the CTQ away from $\mathcal{T}$ and (b) The data item is changing very slowly	1.15
β	Factor by which the $\mathcal{C}$ is reduced if the CTQ exceeds $\mathcal{T}$ after a long time	0.75
C_{max}	The maximum value of $\mathcal{C}$	$1
TTR_{min}	The minimum value of TTR	1 second
TTR_{max}	The maximum value of TTR	60 seconds

Table 2: Value of Parameters used in the experiments



(a) For stock 1

(b) For stock 2

Figure 1: CTQ Changes and Variation of $\mathcal{C}$ with Time

We see that in this trace the portfolio exceeds the threshold at the beginning and towards the end of the observed time interval. In the middle part i.e., from time = 600 to time = 2700 the portfolio does not exceed the threshold. Hence the $\mathcal{C}$s of both the stocks gradually reach the C_{max} value of \$1. As the $\mathcal{C}$ reaches the C_{max} value the polling frequency is reduced and this reduces the network overhead. Around time = 2700, when the portfolio exceeds the threshold after a long duration the $\mathcal{C}$ is reduced by the factor β (0.75). As the $\mathcal{C}$ decreases, the polling frequency increases and this helps in tracking the state of the portfolio correctly. If the change in data value is less than the $\mathcal{C}$ (for that stock) then $\mathcal{C}$ will again increase till it reaches the C_{max} value. Thus the $PullCEA$ tries to reduce the network overhead by increasing the $\mathcal{C}$ (which in effect will reduce the polling frequency) when there is a lower chance of the CTQ exceeding the threshold. However, at the same time it does not compromise on the fidelity by reducing the $\mathcal{C}$ once it detects that the CTQ has exceeded the threshold.

In the experiments, the $PullCEA$ detected that the portfolio had exceeded the threshold for about 1159 seconds when the portfolio had actually crossed the threshold for 1016 seconds. This implies that there were a few false positives but they are not as harmful as false negatives.

When the $\mathcal{C}$ is not adjusted dynamically, an additional 101 false positives were observed. The network overhead of the latter was about 971 messages whereas the number of messages for $PullCEA$ were only 551, a gain of about 43% in the network efficiency. Figure 2 shows the comparison of the network overhead incurred by $PullCEA$ and that when $\mathcal{C}$ is not adjusted dynamically. The network overhead has been normalized with respect to the total length of the trace. The graph shows that the $PullCEA$ has around 40% less network overhead. The graphs further reveal that as the threshold value of the portfolio is increased, the network overhead goes on decreasing. As the threshold of the portfolio increases the number of times that the portfolio crosses the threshold decreases. In the $PullCEA$ the number of pulls increases only when the threshold is reached. For the rest of the duration the $PullCEA$ increases the $\mathcal{C}$ of the data items so that the number of pulls is reduced. Hence as the threshold increases the pulls (i.e. the network overhead) decrease due to reduced number of threshold crossings. In case where the $\mathcal{C}$s are allocated statically, the $\mathcal{C}$ allocated to each of the data item increases with the increase in the threshold. With an increase in the $\mathcal{C}$ the value of TTR calculated by the *Adaptive TTR Algorithm* increases and hence the network overhead reduces to some extent. But the added decrease observed with $PullCEA$ is not observed here. Since the static algorithm does not consider the dynamics hence it is not as efficient as the $PullCEA$. The $PullCEA$ was also evaluated with more data items per CTQ and the results observed were consistent with that seen with a CTQ of two stocks.

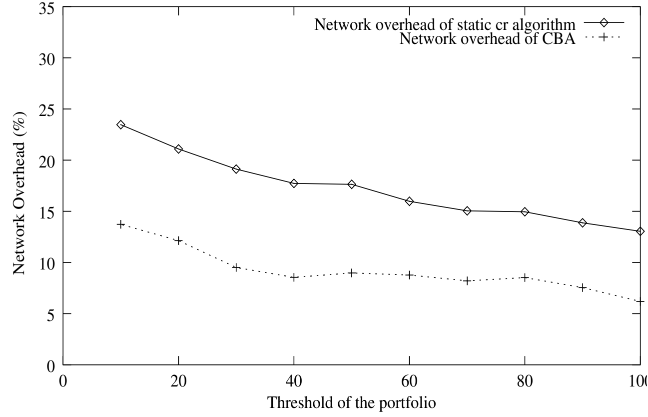
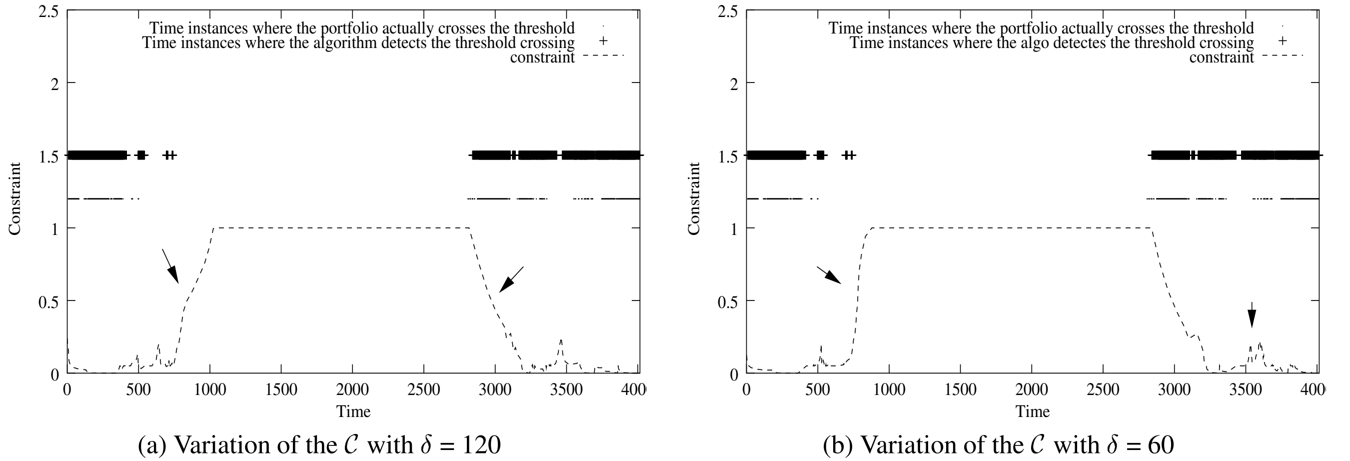


Figure 2: Comparison of the network overhead of $PullCEA$ and static $\mathcal{C}$ Algorithm

2.6 Effect of varying the parameters of $PullCEA$

The $PullCEA$ provides several tunable parameters, namely δ, β and γ that can be used to control the algorithms behavior. This section illustrates the effects of varying these parameters on the fidelity offered by the algorithm and on the network overhead.

If the portfolio value exceeds the threshold after remaining in the same state for δ time then the $\mathcal{C}$ is pulled down by factor β. If the factor δ is increased then a larger amount of history will influence the $\mathcal{C}$ and the $\mathcal{C}$ will be decreased more often. The figure 3 shows the comparison of the variation of the $\mathcal{C}$ for two different values of the δ. As can be seen in the figure 3(a) with larger δ, the value of $\mathcal{C}$ increases gradually (observe the $\mathcal{C}$ values at the first two arrows). In Figure 3(b) the $\mathcal{C}$ value increases more often (notice the $\mathcal{C}$ value near the second arrow). Thus with an increase in the value of δ the fidelity increases at the expense of larger network overhead.



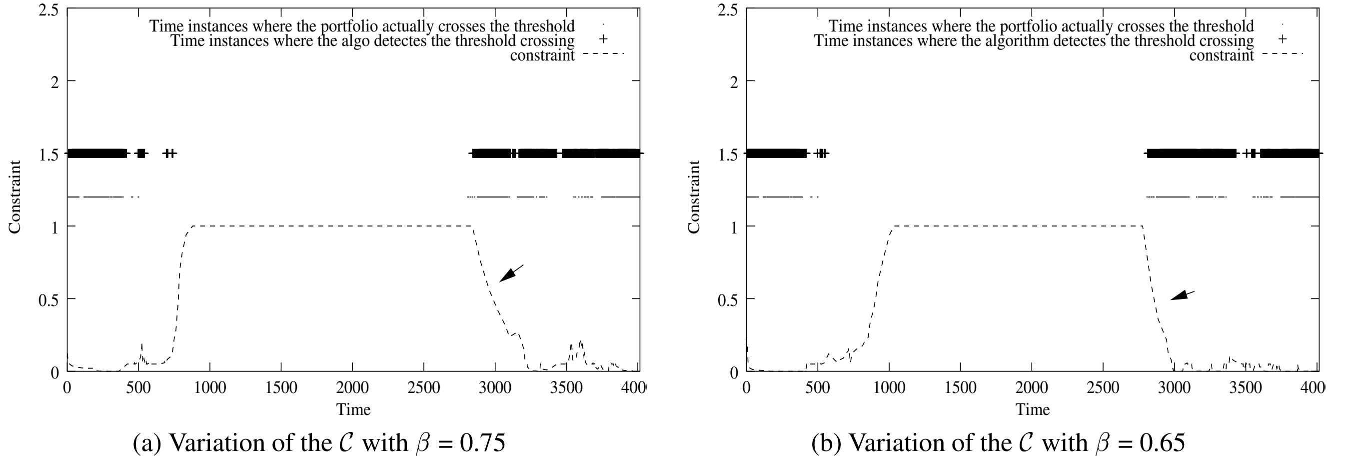
(a) Variation of the $\mathcal{C}$ with δ = 120

(b) Variation of the $\mathcal{C}$ with δ = 60

Figure 3: Effect of change in δ on $\mathcal{C}$

If the portfolio exceeds the threshold after a long duration the $\mathcal{C}$ is decreased by the factor β. The portfolio can exceed the threshold by either (a) a sudden momentary change in the data value of any of the stocks or (b) a change in the data value that persists for a long duration. If the polling is infrequent then the first case will lead to false positives. The factor β helps to avoid this. If the value of β is reduced then the $\mathcal{C}$ will reduce by a larger amount and the chances of false positives will reduce. But if the change in the data value persists for a long duration then an increase in the value of β will lead to unnecessary polling resulting in an increase in the network overhead. Figure 4 shows the effect of decreasing the value of β. A comparison of the two graphs reveals that in the figure 4(b) (with

smaller β) when the portfolio threshold is exceeded there is a sharp decrease in the $\mathcal{C}$ (compare the $\mathcal{C}$ values near the arrow). It was observed that in the second case when the β was set to 0.65 the number of false positives decreased sharply.



(a) Variation of the $\mathcal{C}$ with β = 0.75

(b) Variation of the $\mathcal{C}$ with β = 0.65

Figure 4: Effect of variation of β on $\mathcal{C}$

Having examined a pull-based approach to executing CTQs, we now move on to a push-based approach.

3 Push-Based CTQ Execution Approach ($PushCEA$)

In this section we present and evaluate *PushCEA*. For reasons that will become clear shortly, in addition to using a push-based approach, here, $\mathcal{C}$s are represented in terms of coherency windows. A coherency window denotes the upper and lower limit within which the value of data item may vary at the server without being updated to the proxy.

3.1 Algorithm

The algorithm is made up of the following steps:

- Initially each proxy calculates the coherency window for every data item depending on its contribution to the CTQ as there is no history corresponding to the dynamics of the data. These coherency windows are then informed to the corresponding sources. Whenever the value of a data item moves outside its coherency window at the server, the server pushes the new value to the proxy. Section 3.1.1 explains this.
- When a new data value is pushed by a source to a proxy, the proxy adjusts its coherency window.This is described in section 3.1.2 and does a fidelity check to ensure that the CTQ's threshold specifications are met.

Let w_i denote the coherency window of ith data item d_i of the CTQ:

$$w_i = < l_i, u_i >$$

where l_i is the lower end of w_i and u_i is the upper end of w_i. Thus if p_i is the value of the data i at the proxy, $l_i \leq p_i < u_i$.

3.1.1 Initial allocation of coherency windows to data items

Given a coherency window, w_i, its l_i is initialised as (p_i - c_i) and u_i is initialised as (p_i + c_i) where c_i is calculated as in Section 2.1. The w_i of each data item is adapted dynamically as the value of the data changes. How we achieve this, is discussed next.

Table 3: Parameters used in $PushCEA$

Parameter	Purpose	Value
α	Determines the extent by which a $\mathcal{C}$ should be increased if the new data value takes the CTQ in a direction away from the threshold	1.5 - 2.5
$\alpha^{'}$	Determines the extent by which $\mathcal{C}$ should be changed if the new data value takes a CTQ in a direction towards the threshold	0.5 - 1.5
σ	Used to determine fidelity. If it is close to 1, it leads to high fidelity	1 - 2.5 1 for maximum fidelity

3.1.2 Dynamic redistribution of w_i

Whenever the value of a data at its source is updated and is not within its w_i, the source pushes the new value to the proxy. The recency in changes of the value of the data item signifies that the coherency window allocated to the data item is not large enough to cover the variations in its value and hence needs to be given a larger window. This will help in reducing the network overhead. However arbitrary increase in the coherency window is constrained by the need to maintain fidelity. This problem can be analysed in the form of following cases: if the value of the CTQ (a) moves away from the threshold (case 1),(b) moves towards the threshold and does not cross it (case 2) and (c) moves towards the threshold and also crosses it (case 3). It is evident that the above cases cover all the possibilities.

1. When the CTQ moves away from the threshold (Case 1):

   If the new value of the data moves the CTQ value away from the threshold (as shown in fig 5) the w_i is increased symmetrically. This will happen if

   - The initial value of the CTQ was less than the threshold and the change in the data value reduces the value of the CTQ further or
   - The initial value of CTQ was more than threshold and the change in data value has increased the value of CTQ further.

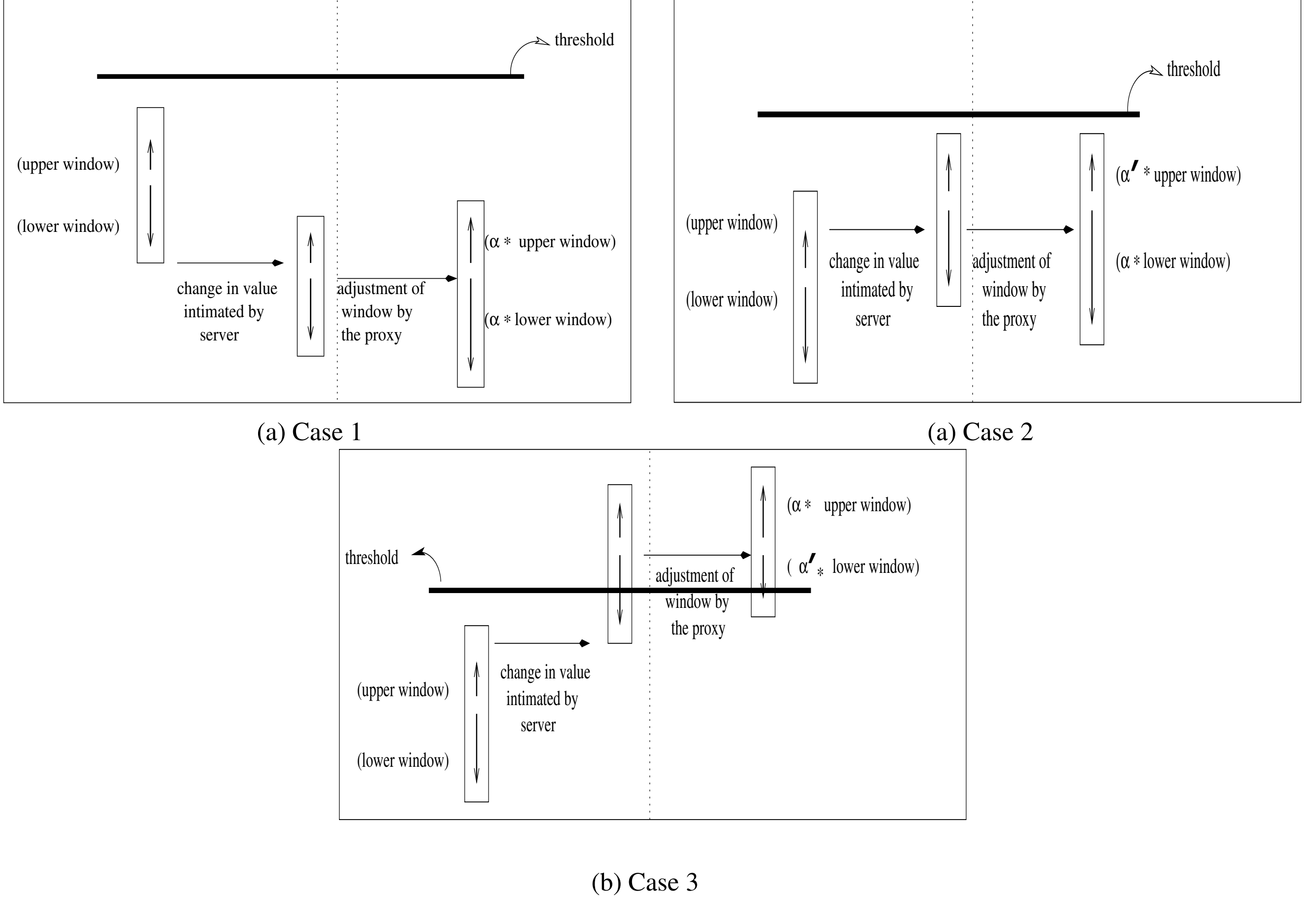
   If the w_i was initially $< l_i, u_i >$, it is changed to $< \alpha * l_i, \alpha * u_i >$. The client is interested in knowing only the correct state of the CTQ. In the above two cases the client is correctly informed about the state of the CTQ. As the data values are taking the CTQ away from changing its state, the source can afford to be less accurate. Hence we increase the w_i size to reduce the network overhead.

2. The CTQ value moves towards the threshold (Case 2):

   If the new data value moves the CTQ towards the threshold the w_i is not increased symmetrically.

   If the most dynamic data (the one that has most recently changed) has changed to bring the CTQ close to the threshold (as in fig 5), we can't afford to increase its w_i. Initially if the w_i was $< l_i, u_i >$ and p_i has increased to bring the CTQ closer to the threshold, the new w_i is $< \alpha * l_i, \alpha^{'} * u_i >$ where $\alpha^{'}$ is less than α so as to maintain more accuracy in the upper limit of the data value. Similarly if the w_i was $< l_i, u_i >$ and p_i has decreased to bring the CTQ closer to the threshold, the new w_i is $< \alpha^{'} * l_i, \alpha * u_i >$. The effect of parameters α and $\alpha^{'}$ have been experimentally determined and the performance of the algorithm with variation in their values is discussed in section 3.2.

3. The CTQ value crosses the threshold (Case 3): If the CTQ value crosses the threshold as a consequence of the most recent change (as in fig 5) we stretch the w_i so that the bound that is now closer to the threshold, is



(a) Case 1

(a) Case 2

(b) Case 3

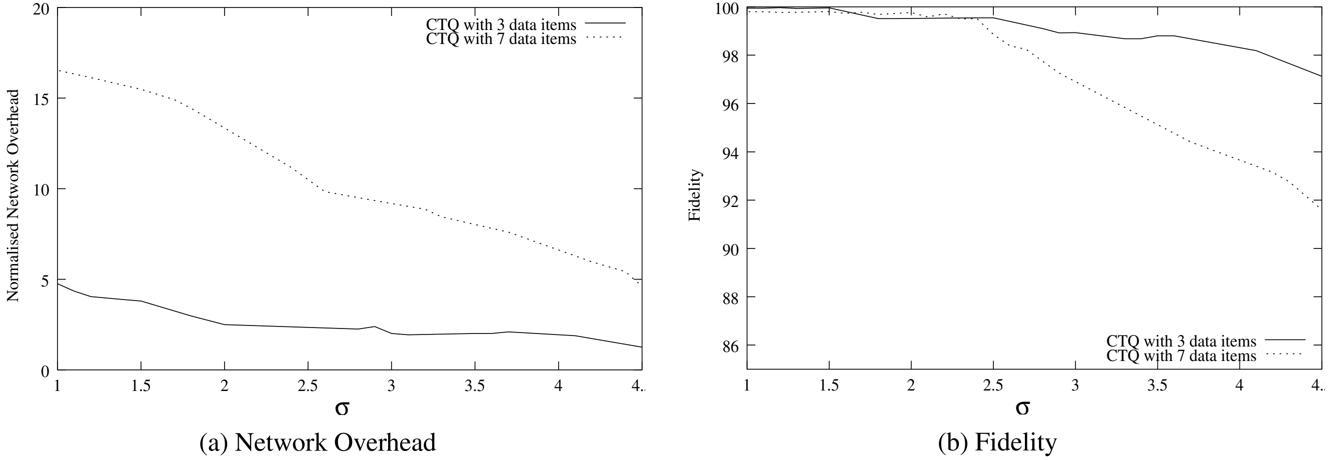
Figure 5: Different Cases for the Dynamic Redistribution of w_i

stretched by α' and the one that is now away from the threshold is stretched by α. Note that in this case we swap α and α' with respect to **case 2** because now the other bound is closer to the threshold. For instance, if the initial w_i was $< l_i, u_i >$ and p_i has increased to make the CTQ cross the threshold, the new w_i is $< \alpha' * l_i , \alpha * u_i >$. Since α' is less than α we maintain more accuracy near the threshold. Similarly if the w_i was $< l_i, u_i >$ and p_i has decreased to bring the CTQ closer to the threshold, the new w_i is $< \alpha * l_i , \alpha' * u_i >$.

Note that the new coherency windows are determined without taking the CTQ's threshold specification. So, before conveying the the recomputed window values to the sources, the proxy checks to make sure that CTQ value is not above the threshold. If it is, then it chooses one or more data items and modifies their coherency windows so that with the resulting windows, 100% fidelity is assured. The chosen data items are the largest contributors to the excess in the CTQ value beyond the threshold.

Note that this 100% fidelity is ensured even with the worst case value of a data item, i.e., when the data item has a value equal to one of the the extremum values of the window. This observation provides us some leeway in trading off fidelity for network overheads. Specifically, even if we allow some coherency windows to be slightly larger than is needed to maintain 100% fidelity, we may yet achieve close to 100% CTQ fidelity. This is achieved by using a factor σ (≥ 1) such that the effective *CTQ threshold value* is set to σ times the specified value and the coherency windows are determined (as discussed above) with this new relaxed threshold value. In Section 3.2 we study the effect of σ on fidelity and overheads and show that we can achieve a large drop in network overheads (and server load) without any significant drop in fidelity by using this simple technique.

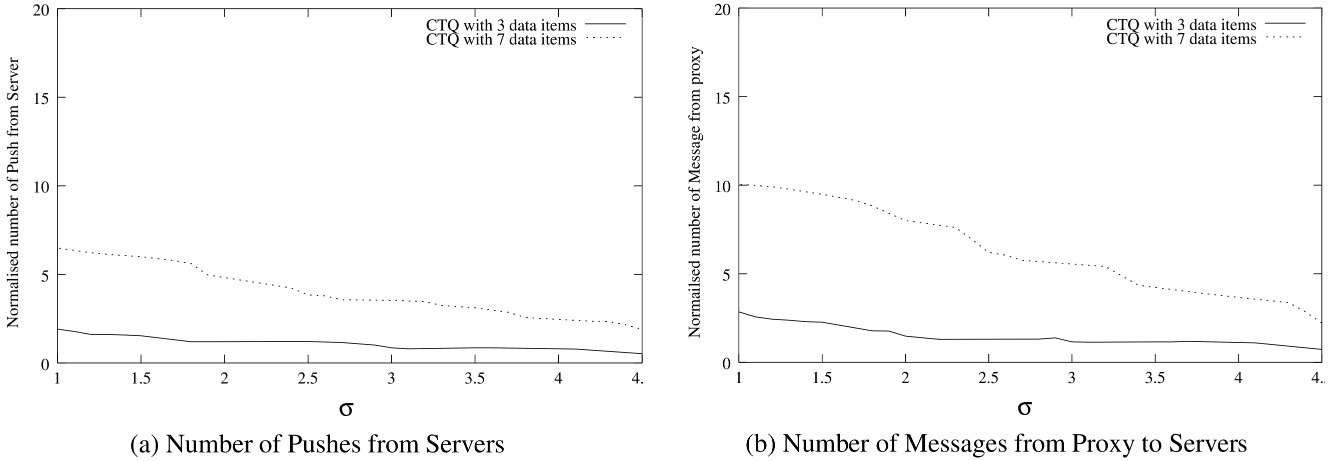
3.2 Analysis of the $PushCEA$



(a) Network Overhead

(b) Fidelity

Figure 6: Variation with size of CTQ



(a) Number of Pushes from Servers

(b) Number of Messages from Proxy to Servers

Figure 7: Nature of Network Overheads

In this section we evaluate the performance of $PushCEA$. Fig 6 shows the fidelity and network overheads of two different CTQs, one using 7 data items and the other 3 data items, under changing σ. From Fig 6(b) we see that $PushCEA$ can achieve very high fidelity for smaller values of σ upto 1.5. However, in terms of network overhead, the algorithm works wells only for small CTQ's. As we increase the size of the CTQ, the network overhead for the same fidelity level increases. Fig 7(a) explains why this is so. With an increase in number of stocks, the number of pushes increase. Consequently, for every push from server the proxy recalculates the coherency windows for the stocks and informs the new windows to the server. The number of coherency windows that need to be recalculated by the proxy depends on the strategy explained above. From Fig 7(a) and 7(b), we observe that the ratio of the number of messages from proxy to server and push messages from the server suffers a marginal increase with increase in the size of the CTQ. This ratio signifies the number of coherency windows that need to be recalculated at the proxy for every push from the server. This ratio is between 1 to 2 due to proper selection of data items whose coherency window is decreased during fidelity check. In case the stocks are selected randomly, this ratio could be much greater than 1 to 2.

This approach has an edge over the pull based approach because of the following:

- It can be tuned to achieve very high fidelity as per individual needs. Values of σ closer to **1** lead to high fidelity. In case of network bottlenecks, we can use higher values of σ and yet obtain close to 100% fidelity.
- It can be deployed over a multiple layer network where there are multiple proxies, one proxy serving the other. If $PullCEA$ is deployed in such a network, the fidelity experienced by an end user at level "n" will be $92 \times (0.92)^n\%$ assuming that fidelity of $PullCEA$ is 92%. Whereas in case of $PushCEA$, which can achieve fidelity as high as 99%, the corresponding fidelity will be $99 \times (0.99)^n\%$ which is much higher.
- This approach takes care of abrupt changes in the price of a stock at the server because these will be notified to the proxy if the latest price of a stock does not lie within its coherency window.

However this approach is not scalable. If the CTQ's data set is very large, the network overhead increases. To overcome the scalability problem, the proxy can resort to the use of pull based service for some of the data items and push for the rest. This leads to the idea of combining pull and push based approaches for the efficient execution of CTQs.

4 A Hybrid CTQ Execution Approach (HyCEA)

A simple way to combine the $PullCEA$ with the $PushCEA$ is to split the CTQ's data items into two parts. The $PullCEA$ will then have a fraction of the threshold corresponding to data items allocated to it. But the problem with this approach is that it can lead to unnecessary network overhead. Consider a case when the value of the data items using the $PushCEA$ is moving toward the threshold. Due to this the coherency windows will reduce and a lot of changes in the data value will be notified to the proxy. At the same time if the value of the data items using $PullCEA$ moves in the opposite direction then as a whole the CTQ will not exceed the threshold. Hence the advantage of considering the data items constituting the CTQ as a semantic unit is lost because unnecessarily the data items using $PushCEA$ incur overheads. Hence a better approach is needed. Such an approach is the $HyCEA$.

4.1 Calculation of Coherency Constraints for $HyCEA$

For the $PushCEA$, α and α' are instrumental in shaping the coherency window. Whereas for the $PullCEA$, β and γ are instrumental in shaping the $\mathcal{C}$. In $PullCEA$, β is used to reduce the $\mathcal{C}$ of the data item if a change in its value causes the CTQ to cross the threshold. This job is quite similar to that served by α' in $PushCEA$ as it prevents the coherency window from increasing inspite of a change in the data item value because the CTQ is close to the threshold. Similarly we can draw a parallel between α and γ as they both help in increasing the coherency window (or $\mathcal{C}$ in case of pull) in case of increased changes in the value of the data item. This relation can be expressed as follows:

$$\alpha = \text{(some scaling factor)} \times \gamma$$
$$\alpha' = \text{(some scaling factor)} \times \beta$$

Let us see what happens if we keep this scaling factor as unity. In this case, we can calculate the $\mathcal{C}$s and coherency windows directly from each other. Whenever there is a change in $\mathcal{C}$ or the coherency window we calculate the other from it.

- for a change in the $\mathcal{C}$ of a data item under $PullCEA$ we assign it a new coherency window as $< p_i - \mathcal{C}, p_i + \mathcal{C} >$ where p_i is the data value for the ith data item at the proxy.
- for a change in the coherency window of a data item under $PushCEA$ we assign $\mathcal{C}$ as half of the coherency window.

Thus in $HyCEA$ the coherency window of a data item using $PushCEA$ is changed under the following circumstances

- The data value at the server moves out of the coherency window thereby increasing the interval
- It may be changed by the proxy
- If the $\mathcal{C}$ of the data item (calculated by the $PullCEA$) changes then this change is reflected, and the coherency window is changed accordingly.

The $\mathcal{C}$ of a data item using $PullCEA$ can be changed, if the coherency window assigned to it by the $PushCEA$ changes. Thus there is a perfect blend of the two approaches.

The *Adaptive TTR Algorithm* places an upper limit on the TTR. This ensures that the data items that use $PullCEA$ will be out-of-sync with the server by maximum TTR_{max} seconds. Such a limit is also required for the data items using $PushCEA$. Consider a scenario where the CTQ value is away from the threshold. The $PullCEA$ will increase the $\mathcal{C}$ value to reduce the network overhead. If this $\mathcal{C}$ influences the coherency window then the window so obtained will be very large. In the absence of any upper limit on the inter-push duration, the data value may never move out of the coherency window. This will have an adverse effect on the fidelity when the value of the CTQ moves near the threshold. Hence to prevent fidelity reduction and to increase resiliency, we set a maximum limit on the duration for which a proxy may not be informed about the latest value of a data item. In case this limit $\mathcal{L}$ is reached, the server forces a data value to the proxy. This results in recalculation of the coherency windows and thus better coherency results.

4.2 Distribution of data items between $PullCEA$ and $PushCEA$

The Push connections will have a higher fidelity and hence it makes sense to give this service to the data items with maximum fluctuation (hot data items). This helps in reducing the network overhead as the hot data items have a greater contribution to the network overhead. In the simplest approach the allocation of the data items that use $PushCEA$ can be done statically depending on the contribution of the data item to the entire CTQ; those data items that have a higher initial value or whose number of data items are larger have a greater influence on the CTQ. Such data items should be maintained more accurately and hence are given push connections. But the main drawback in this method is that it does not consider the dynamics of the data. It can happen that initially the value of a data item may be larger but later due to fluctuations the value may drop resulting in the contribution of the data item getting reduced. Hence there is a need to periodically adjust the data items that use $PushCEA$ to cater to the varying dynamics of the data. The proxy can keep track of the changes during a learning period. $PullCEA$ calculates the $\mathcal{C}$'s of all the data items. The proxy keeps track of the average $\mathcal{C}$'s for all the data items during the learning period. At the end of this period those data items whose $\mathcal{C}$ is below the threshold defined by

$$C_{min} + ppc \times (C_{max} - C_{min})$$

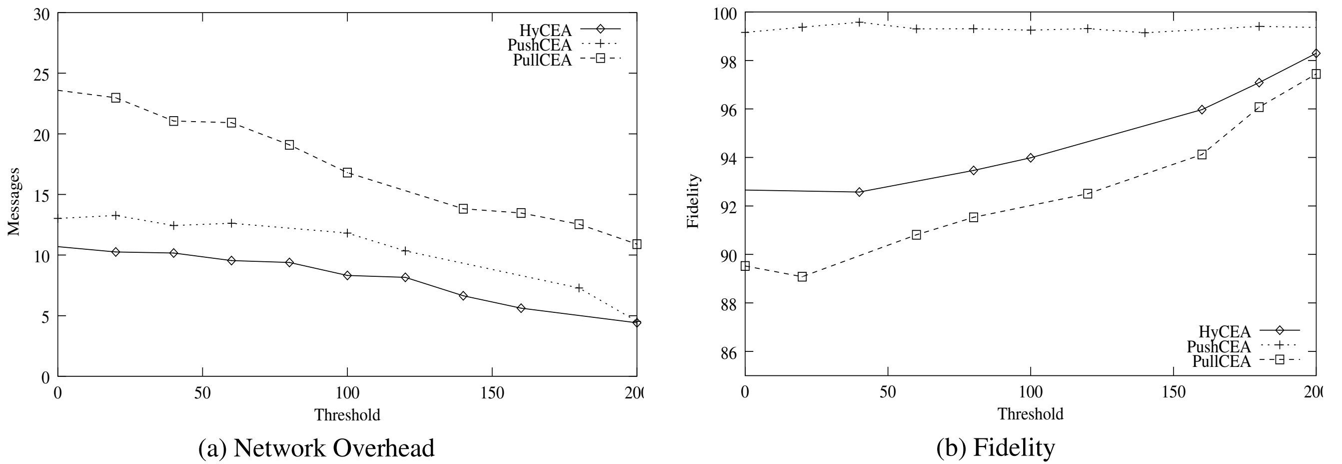
where C_{min} is the minimum $\mathcal{C}$ seen during the evaluation period, C_{max} is the maximum $\mathcal{C}$ seen during the evaluation period and *ppc* is the percentage of push connections for the CTQ.

If none of the data items of the CTQ have $\mathcal{C}$ below this threshold then all the data items will get a pull connection thereby saving a few push connections which can now be allocated to some other more deserving data items of the same proxy (but for a different CTQ). Thus this approach adapts to the dynamics of the data, but it requires additional history of the data items to be maintained at the proxy.

4.3 Experimental results

In this section we evaluate the different approaches and show how each approach fares in terms of fidelity and network overhead. The x-axis of the graph signifies the offset from the median. There were 7 to 8 data items in the CTQ's used for these experiments.

Figure 8 compares the fidelity and network overhead of the three algorithms namely $PullCEA$, $PushCEA$ and the $HyCEA$. From figure 8 it is evident that the $HyCEA$ is able to maintain the network overhead to minimum amongst all the approaches. In $PushCEA$, because of the state information at the server none of the interesting changes are missed by the proxy. Hence the fidelity offered by $PushCEA$ is maximum. $HyCEA$ misses out on some interesting changes, resulting in decreased fidelity than $PushCEA$. Because of these missed changes the network overhead of $HyCEA$ is the least. The other extreme to this approach is $PullCEA$ which does a lot of unnecessary pollings, as is evident from the high network overhead. With an increase in the threshold, the network overhead of $HyCEA$ is same as that offered by $PushCEA$ and the fidelity offered is also comparable to $PushCEA$. Hence for large threshold values $HyCEA$ is most resilient approach and offers very good fidelity and network characteristics. In all the curves it is observed that as the value of the threshold increases the fidelity goes on increasing. This can be attributed to the fact that with an increase in the threshold value the number of times that the CTQ actually crosses the threshold decreases. Hence all the algorithms are able to judge the state of the CTQ in a better way. If the threshold is nearer to the median then the chances of the proxy being mis-informed are more. Hence the coherency windows of the data items will be less and the $\mathcal{C}$'s of the pull data items will be reduced by $PullCEA$. Smaller coherency windows and $\mathcal{C}$'s result in larger messages and hence the network overhead is maximum at the median.

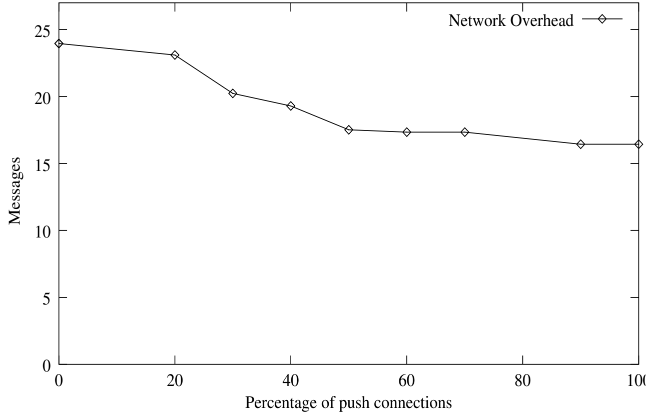


(a) Network Overhead (b) Fidelity

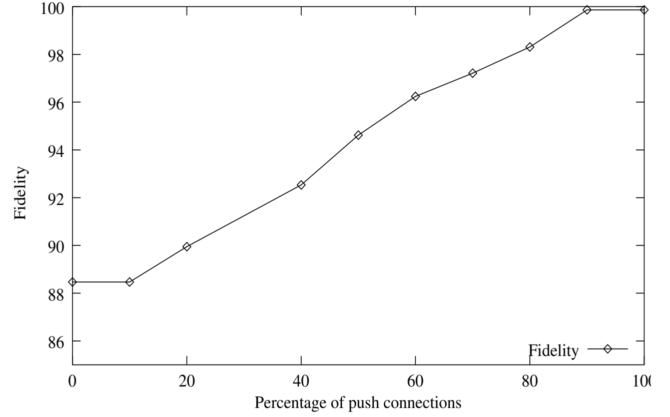
Figure 8: Effect of Threshold Parameter

Figure 9 portrays the change in fidelity and network overhead with an increase in the number of push connections. As the percentage of push connections is increased the proxy is able to judge the state of the portfolio more accurately and hence the fidelity increases. A push connection sends data to the proxy only when a change in the data value moves it outside the coherency window. On the other hand the pull connection has to estimate the data value at the server and hence can have a lot of unnecessary messages. As a result the network overhead decreases as the number of push connections is increased.

Figure 10 shows an interesting result about the network overhead and fidelity as the value of the L is changed. The graphs show that there is little change in the network overhead with an increase in the value of the L, but the fidelity goes on decreasing. As the value of the L is increased, the server sends periodic messages to the proxy after larger intervals of time. This increases the chances of the proxy being uninformed for a longer period thereby causing more fidelity violations.

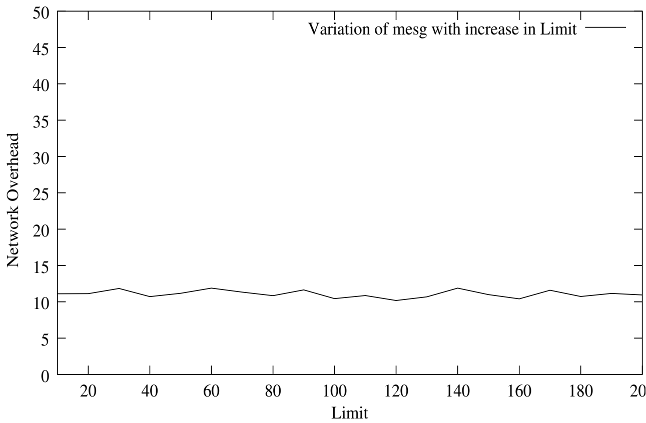


(a) Variation of Messages

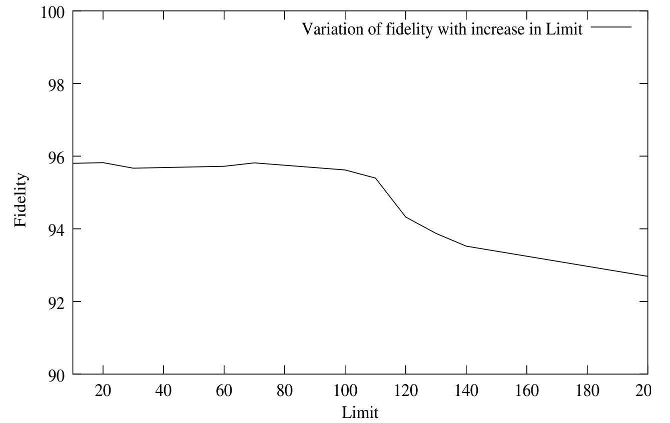


(b) Variation of Fidelity

Figure 9: Behaviour of the algorithm with increasing push connections



(a) Network Overhead



(b) Fidelity

Figure 10: Performance of the $HyCEA$ under different $\mathcal{L}$

5 Related Work

The problem of consistency maintenance between a data source and cached copies of the data was first studied in [2]. The paper discusses techniques where a source pushes updates to clients based on expiration times associated with the data. Since then numerous efforts have investigated push-based dissemination techniques [1, 3, 5, 4, 8, 13], however none of these techniques explicitly target time-varying data with associated coherency constraints.

More recently, the problem of consistency maintenance has been studied in the context of web caching and several techniques such as client polling [7], adaptive time-to-live (TTL) values [17], server invalidation [11] and leases [18, 24] have been proposed. These efforts typically assume that cached data is modified on slow time scales (e.g., tens of minutes or hours) and are less effective at maintaining consistency of rapidly changing data cached at proxies. Caching of *dynamic* content has been studied in [10] wherein a scheme based on push-based invalidation and dependence graphs is proposed, while another effort has focused on availability and scalability by adjusting coherency requirement of data items [9]. Neither effort has explicitly addressed coherency maintenance for efficiently executing queries at a proxy.

Mechanisms for disseminating fast changing documents are proposed in [16, 15, 14]. The difference between these approaches and ours is that they disseminate *all* updates to the document using Multicast, while we selectively disseminate updates based on the coherency requirements of a data item. The concept of approximate data at the

clients has studied in the context of stock price dissemination [25]; the approach focuses on individual stocks and does not address the additional mechanisms necessary to track a portfolio of stocks.

Finally, our CTQs are a subset of the general class of continuous queries over dynamically changing data [12, 6]. Continuous queries in the Conquer system [12] are tailored for heterogeneous data, rather than for real time data, and uses a disk-based database as its backend. NiagraCQ [6] focuses on efficient evaluation of queries as opposed to temporally coherent data dissemination to proxies (which in turn can execute the continuous queries). A proxy-based 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 for efficient execution of CTQs at a proxy. Specifically, our approach is proxy-based and (a) dynamically assigns coherency requirements to data items comprising the CTQ, (b) uses adaptive dissemination mechanisms for executing the CTQ with high fidelity and a low cost.

6 Conclusions and Future Work

In this paper, we presented new techniques for monitoring and disseminating dynamic data to proxies where a proxy is interested in executing continuous queries over a group of data items. Our overall approach requires some amount of history to be kept, which can be easily supported by the proxy, as the proxy caters to less number of clients vis-a-vis the server. A useful feature of our techniques is that they have several adjustable parameters which can be used to tune to algorithm to get the desired fidelity and network overhead characteristics. The network overhead offered by the $PullCEA$ is around 40% less than that offered by the static $\mathcal{C}$ approach. ?? say something summarizing the other approaches also ?? Our algorithms achieve good fidelity at low cost by considering the data items in a CTQ as a semantic unit, since this reduces the dissemination overheads of all the data items needed for a CTQ if the data values are changing such that there is very little chance of a CTQ's threshold being exceeded..

CTQs queries can also be implemented by using other techniques such as PAP [20] and leases[23, 24]. Exploring these possibilities is left as future work.

In practice a server will cater to many proxies, some of them catering to the same stock. This may result in multiple coherency windows for the same stock being kept at the server. Another variation that is possible to our approach is to keep track of the most precise approximation over all the CTQ's instead of individual CTQ's. In such a case any violation in the value of the data at the server with respect to its minimum window forces the server to broadcast the new value to all the proxies resulting in a network overhead of the order of **O(m)** where m is the average number of proxies for each data item. A subsequent check by each proxy will cost **O(m)** computation (0(1) for each proxy). In our case the state space required at the server is **0(m)** but the network overhead is reduced to only those which require this data value..

References

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

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

[3] 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.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing System*, 1999.

[5] 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.

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

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

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

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

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

[12] 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.

[13] 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.

[14] 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.

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

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

[17] 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.

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

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

[20] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, Adaptive Push-Pull: Dissemination of Dynamic Web Data *10th International World Wide Web Conference , Hong Kong, May 2001.*

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

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

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

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

[25] C. Olston, B. T. Loo, and J. Widom Adaptive precision setting for Cached Approximate Values. In *Proceedings of the ACM SIGMOD Conference*, May 2001.