Temporally Consistent Delivery of Time-Sensitive Information: Solutions and Challenges

Krithi Ramamritham

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai, India 400076
krithi@cse.iitb.ernet.in

Abstract

The web and the internet have become the dominant infrastructures for information access and dissemination. Even time varying information such as currency and stock prices, real-time data needed for tele medicine, etc., are being delivered via the Web. Clearly, the Quality of Service needs for delivering such time-critical data are much more stringent than for static Web pages. This paper examines the nature of the dynamics on the Web, studies the suitability of the current infrastructure for disseminating time varying information, and discusses fresh approaches tailored for dynamic data. A challenging and fruitful research direction is to examine the use of a control theoretic framework to address the problem of tuning the algorithms needed for adaptive data delivery. *copyright IFAC2001*

Keywords: dynamic data, data dissemination, consistency, continuous queries, quality of service.

I. Introduction

An ever increasing fraction of the data delivered using today's internet resources is time-varying (i.e., changes frequently). Examples of such data include sports information, news, and financial information such as stock prices. Tele medicine is another application for which the web is used. An important issue in the dissemination of time varying data in such applications,i is the maintenance of temporal coherency. The coherency requirements on a data item depends on the nature of the item and user tolerances. To illustrate, a user may be willing to receive sports and news information that may be out-of-sync by a few minutes with respect to the server, but may desire stronger coherency requirements on the status of patients' critical parameters.

The World Wide Web itself can be viewed as a global virtual database. When time varying data is stored in this database, it is not only necessary to keep this virtual database consistent with the state of the real world that it is supposed to capture, but also essential to transmit the contents of this virtual database to the consumers of the data so that the discrepancies between what the consumers see and what the real world's state is, is within acceptable tolerance. In other words, the temporal quality of the data viewed by consumers should be acceptable. We call this requirement *temporal consistency maintenance*.

On the Web, the sources of data are autonomous with no regulations on the way information is represented. The data that we have to deal with is of a temporal nature, characterized by having a) rapid and b) unpredictable changes. Given a choice between the methods of obtaining data, the system must be capable of choosing the right method depending on the prevailing conditions, the major factors in the consideration being server capabilities, user tolerance and the network bandwidth available. At the same time, the user must not be made to request the same data again and again, but should be able to run continual queries on this dynamic data.

Given these challenges, we have been working towards a system with the following goals:

- It should be capable of dealing with Dynamic Data which, as we mentioned before changes rapidly and unpredictably.
- It should be able to retrieve data from multiple sources.
- It should provide high fidelity to the users. We define fidelity as the percentage of time for which user's requirements are met.
- It should be scalable with respect to the number of users, network capacity, computations, state space.
- It should be tolerant to faults at the data sources.
- It should forward only relevant information to the users.

In this paper, we discuss the progress made till now in meeting these requirements and also discuss the challenges that remain.

We start with the considerations involved in the dissemination of individual data items to understand the scope of the problem and the possible solutions. Then we examine the problem when several data items, e.g., several stock prices, need to be viewed together, introducing the need to address the issue of mutual consistency. Finally, we extend the problem domain to consider the issues arising from the need to execute arbitrary continuous queries, i.e., long-running queries, over the virtual database. We end the paper with a discussion of the challenges that remain, including a formal, perhaps control-theoretic, framework to dynamically determine the tunable parameters and to make adaptive decisions.

II. Disseminating Individual Dynamic Data

To maintain coherency of the cached data, data as seen by a user must be periodically refreshed with the copy at the server. We assume that a temporal coherency requirement tcr is attached to each cached item of interest. The value of tcr denotes the maximum permissible deviation of the cached value from the value at the server and thus constitutes the user-specified tolerance. Let $S(t)$ and $U(t)$ denote the value of the data item at the server and the user, respectively. Then, to maintain temporal coherency (tc) we should have

$$|U(t) - S(t)| \leq c.$$

In addition to specifying the coherency requirement c, users can also specify their fidelity requirement f for each data item so that an algorithm that is capable of handling users' fidelity requirements (as well as tcrs) can adapt to users' fidelity needs. In the following sections we explain how one can use user specified constraints to offer high fidelity with efficient use of available resources.

A. Client Pull vs. Server Push

The *Pull* approach is the most traditional approach for maintaining the temporal coherency of dynamic data. In this model, each data item is assigned a certain *time-to-refresh (TTR)* when the data object is brought to a data store at the user's end, or cache. Since the arrival of the data item and until time equal to TTR elapses, all the requests for the data object are satisfied from the cache without looking up the values in the data sources. Thus in this approach, the user is responsible for obtaining the data from the server, by issuing a `GET` request to the server and the server just delivers the required data.

Many current applications do *WebCasting* [11]. Users periodically poll the server for this data. This approach is equivalent to setting the *TTR* value of a cached item *statically*. However, this can lead to a very high network overheads in case the polling period is too low or may cause the user to miss some changes of interest if the polling period is too high. Clearly, this technique is useful only if rate of change of data is constant or relatively low (such as news). If the rate of change itself is varying, then this technique of assigning frequency apriori is not suitable (as in the case of stock quotes). But still, currently this is the most popular data delivery technique as it can be purely web-based (because of HTTP) and does not need any special resources (like push capability in servers or modification of HTTP).

Since dynamic data changes independently and unpredictably, we cannot use standard prediction and forecasting algorithms for predicting the next TTR value to be assigned to the data object. We have developed a novel method for assigning TTR values in [9]. Given a user's coherency requirement, this technique allows a proxy to adaptively vary the TTR value based on the rate of change of the data item. The TTR decreases dynamically when a data item starts changing rapidly and increases when a hot data item becomes cold. To achieve this objective, the *Adaptive TTR* approach takes into account (a) static bounds so that TTR values are not set too high or too low, (b) the most rapid changes that have occurred so far and (c) most recent changes to the polled data.

In the *Server Push* method, the server is responsible for delivering the relevant data to the user. The server does not behave in *request-response* mode, where the server delivers some data only when there is an explicit request for it. But instead, server pushes the data into channel without any explicit request. As before, the server can push the data either periodically or aperiodically.

In *periodic push*, the server is not aware of the exact coherency needs of a particular client, but only of the general demand for data items. So, based on the general demand distribution of data items, the server creates a schedule for the periodic dissemination of data items. A data item with higher demand will be disseminated with higher frequency and vice-versa. All data items get divided into frequency bands, where data items belonging to one frequency band have similar demands. Once the push schedule is created using these frequency bands, it is not changed. The server then repeats this schedule periodically. The Broadcast Disks [1] approach is one such approach where the frequency bands are termed as broadcast disks. This approach also provides for client feedback. An interesting property of this approach is that it treats the channel like a medium and tries to decide on the "format" in which the channel should hold the data.

It is clear that since the server is not aware of the specific needs of the users, the push schedule may not be adequate for users who desire high fidelity and specific results. So, the approach may lead to low fidelity and/or wastage of bandwidth. But, the approach is useful for data which is generic and changes not very rapidly (e.g. news, digests, entertainment).

This basic idea has been extended to adaptively disseminate data in asymmetric communication environments in [6] in such a way as to take into account the temporal properties of the data as well as users' deadline requirements. More generally, in the *aperiodic push* approach, a user registers with a server, identifying the data of interest and the associated tcr, i.e., the value c. Whenever the value of the data changes, the server uses the tcr value c to determine if the new value should be pushed to the user; only those changes that are of interest to the user (based on the tcr) are actually pushed. To achieve this objective, the server needs to maintain state information consisting of a list of users interested in each data item, the tcr of each user and the last update sent to each user.

The key advantage of this approach is that it can meet stringent coherency requirements—since the server is aware of every change, it can precisely determine which changes to push and when. A limitation of push-based servers is that the amount of state that needs to be maintained can be large, especially for popular data items. A server can optimize the state space overhead by combining requests from all proxies with identical tcrs into a single request; all proxies are notified if the change to the data item D exceeds a specified tcr. Even with such optimizations, the state space overhead can be excessive, which in turn limits the scalability of the server. A further limitation of the approach is that it is not resilient to failures. The state information is lost if the server fails and requires the proxy to detect the failure and re-register its tcr for the data item.

Thus, the Push and Pull approaches have complementary properties with respect to fidelity, network utilization, scalability and resiliency [4]:

- A pull-based approach does not offer high fidelity when the data changes rapidly or when the coherency requirements are stringent (i.e., small values of c). Moreover, the pull-based approach imposes a large communication overhead (in terms of the number of messages exchanged) when the number of clients is large. But it may suffice for requests which have large values of c.
- A push-based algorithm can offer high fidelity for rapidly changing data and/or stringent coherency requirements. However, it incurs a significant computational and state-space overhead resulting from a large number of open push connections and their serving processes/threads. Moreover, the approach is less resilient to failures due to its stateful nature.

These properties indicate that a push-based approach is suitable when a client expects its coherency requirements to be satisfied with high fidelity, or when the communication overheads are a bottleneck. A pull-based approach is better suited to less frequently changing data or for less stringent coherency requirements, and when resilience to failures is important. The complementary properties of the two approaches indicate the need for having an approach which combines the advantages of both while not suffering from any of their disadvantages.

B. Adaptive Combinations of Push and Pull

As is clear from the previous discussion, neither push nor pull alone is sufficient for efficient dissemination of dynamic data. Since these two techniques have complementary properties, a few attempts have been made in the past to combine these two canonical techniques. *Adaptive leases* [5] and *Volume leases*[7] are two examples. The former is used for maintaining strong cache consistency in the World Wide Web while the latter is used for caches holding a large number of data items. None of these is meant for highly dynamic data. Nor do they take user requirements into account. In this section we describe the Adaptive Leases approach as well as two algorithms, PaP and PoP, that we have developed for better scalability and coherency for dynamic data.

Leases are like contracts given to a lease holder over some property. Whenever some client requests server for a certain document, server returns that document along with a lease. In other words, a server takes the responsibility of informing the client about any changes during the lease period. Once a lease expires, a client must contact the server and renew the lease. Client can use the cached copy while it has a valid lease over the data item. During valid lease period, client remains in push mode and is switched back to pull mode after the lease expires. Thus the client is alternatively served in push and pull modes.

Clearly, pure leases are not very useful for dynamic data. It is very important to choose a good lease period. For a very high value, client remains in push mode for most of the time and scalability problem may arise. On the other hand, for small values the lease renewal cost may prove very high. *Adaptive Leases* try to dynamically adjust the lease duration. The decision is based on many parameters like popularity of the data item, server state space available and network bandwidth available [5].

In the PaP [4] approach, the proxy operates in pull mode using some TTR algorithm, while the server is in push mode and knows the constraint. Using this constraint and user access patterns, the server tries to predict when a client is going to poll next. If it determines that within this predicted time the client is likely to miss a change of interest, it pushes that change to the client. For predicting the client connection times, the server may run the TTR algorithm in parallel with client or use some simpler approximation of it. Given the network delays, a server waits for the client to pull within a small window around the predicted time. So, once the first few changes are intimidated to the client (by pull or push), the rest of the successive changes can be pulled by the client since the client would have appropriately lowered its TTR as a result of the few pushes.

In the ideal case, the fidelity offered will be 100%, but due to synchronization problems and other factors, it will be slightly less. But, it will always be much greater than pull. Because of the pull component the resiliency of the system will be high. And due to the push component, communication overheads will also be low. PaP also provides for fine tuning of its behavior. It has a few parameters which swing it towards more push or more pull, and thus its performance in terms of fidelity and resiliency can be controlled [4].

Another possibility is to divide incoming clients at the server into either push or pull clients and dynamically switch them to one or the other mode [4]. If resources are plentiful, every client is given a push connection irrespective of its fidelity requirements. This ensures that the best fidelity is offered. As more and more clients start requesting the service, resource contention may arise at the server leading to scalability problems. Few clients are then shifted to pull mode. Thus valuable resources are freed and system scales properly. Contrary to this, when resources again become available few high priority clients are switched back to push mode thus ensuring high fidelity.

The most important issue is how to assign priorities to different clients. Few of the possible parameters are the access frequency of each client, temporal coherency requirement, fidelity requirement and network bandwidth available. Clearly no single criterion suffices but collectively they have the potential to offer high average fidelity still keeping the system scalable [4].

In summary, *PaP* achieves resiliency and temporal coherency while PoP achieves scalability by classifying users as per their capabilities and requirements. Now if we replace the push component of *PoP* with *PaP*, then we have achieved all the major objectives, namely resiliency, scalability and coherency in a single algorithm. This new algorithm is *PoPoPaP* i.e. Push or Pull or *PaP*.

III. From Accessing Individual Data Items to Continuously Querying Data from a Domain

Till now we were dealing with systems tailored to specific instances of queries, such as "Tell me when IBM stocks go up by $c". We will now consider a system that gives users the freedom to ask any query on the database. Continual Queries (CWs) allow users to obtain new results from databases without having to issue the same query again and again. CQs are especially useful in an environment like the Web, where the amount of frequently changing information is extremely large.

A continual query (CQ) is defined as a triple (A, T, R) (**A**ction, **T**rigger, Te**r**mination), where A is a normal query written in some traditional query language (like SQL), T is the trigger condition of the CQ and R is the termination condition. A CQ is "installed" when it is first specified. The initial execution of A takes place on the existing data, when a CQ is installed. The result of this query is returned to the user as the first result of the CQ. The subsequent executions of A depend on values of T and R. The subsequent execution of A takes place when the trigger condition T becomes true, provided that the termination condition R is false.

Developing the infrastructure for executing continuous queries on Web data needs several subproblems to be addressed: Consider a continuous query where a user wants to be informed when the net worth of his/her stock portfolio changes by more than a specified threshold. To this end, a user might execute the following query: `show v1=(100 * MSFT + 200 * INTL + 300 * IBM)`

```
for stocks when change(v1) > $500
till end-of-the-month;
```

Executing this query implies obtaining the current prices of MSFT, INTL and IBM from (their respective) web sources

- using one of the approaches mentioned earlier (Pull, PaP, PoP and PoPoPaP, etc.),
- in such a way that the data is mutually consistent, and
- such that users view of stock portfolio is the same as the actual real-world state of the portfolio.

Formally, the following equation should be satisfied to achieve fidelity

$$if \sum_{all\ stocks} |S(t) \times N(t) - S(t^{'}) \times N(t)| \geq threshold$$

then

$$\sum_{all\ stocks} |C(t) \times N(t) - S(t^{'}) \times N(t)| \geq threshold$$

where

- $S(t)$ is the current stock price at the server
- $S(t^{'})$ is the initial stock price at the server
- $C(t)$ is the current stock price at the client
- $N(t)$ is the number of shares of stock in the portfolio
- $threshold$ specifies the coherency requirement for the stock portfolio.

It may happen that due to loss of coherency, the user may perceive the wrong state for his/her portfolio. There are two classes of misperceptions: (i) *false positive*, when a user thinks that the threshold has been exceeded, whereas it is well within the threshold, and (ii) *false negative*, when the portfolio changes have actually exceeded the threshold but the user is unaware of this. In some sense, the latter could be more harmful than the former because in the case of a *false positive* the proxy can always verify the value of the portfolio by polling all the stocks. But no such solution exists in case of false negatives. So it is important to avoid or at least minimize false negatives. We now discuss approaches to achieving these diverse requirements.

As we have seen earlier, the HTTP protocol used in the Web is inherently pull based. Users need to frequently pull the data from the server based on the dynamics of the data and the coherency requirement of the user. The proxy can use the if-modified-since HTTP requests to obtain any changes in the data value at the server. Using push, the number of false positives and false negatives can be made small, barring network and computational delays. But, very few servers provide push support and this service often comes at a price, namely space and computation overheads at the server. Also, the servers have to be re-engineered to deal with portfolio queries. A final reason for using pull based technique is that the push based approach will require cooperation between the servers of the individual data items, and this may not be feasible. For these reasons, our goal is to work within the standard HTTP protocol and hence use pulls intelligently to maintain the coherence of portfolios within user allowed thresholds values.

To achieve this, we must impose a coherency requirement on each of the data items constituting a portfolio, depending on (a) the overall contribution of the data item to the portfolio and (b) the coherency requirement of the portfolio itself. A very naive way to assign crs to data items would be to assign a small (i.e., very stringent) coherency to a data item that makes the greatest contribution to the portfolio (and hence such a data item will be polled more often). But the problem with this approach is that it does not consider the dynamics of the data items. Depending on the way the values of the various data items are changing the crs of each of the data items can be manipulated, thereby saving a lot of network overhead. For example, consider a portfolio consisting of 100 stocks of company A and 400 stocks of company B. Now if the company B stock price hasn't changed for a long time and the company A stock sees a lot of trading and hence its price fluctuates a lot, then the cr of company A should be reduced and that of company B should be increased. This will help in reducing the false negatives and it will also help in saving a lot of unnecessary pulls of the company B stock. Hence there is a need to judiciously assign the cr to each of the data items depending on the dynamics of the data. We have developed an algorithm [2] that continuously changes the cr associated with each of the data item constituting the portfolio based on the dynamics of the data, as well as the relative performance of the data item with respect to the other items in the portfolio. If false positives and false negatives are not to occur then the proxy must present a logically consistent view of the data items cached [10]. Our algorithm achieves this while aiming to reduce the network overhead – by pulling very rarely when there is very little chance of the threshold being exceeded. The algorithm uses the *Adaptive TTR Algorithm* [9] to compute the time after which a data item should be polled so that the coherency associated with that data item is not violated. How to achieve this is the crux of [2], wherein we provide experimental evidence for the efficiency of the approach. The algorithm requires some amount of history to be kept, which can be easily done. A useful feature of our technique is that it has several adjustable parameters which can be used to tune the algorithm to get the desired fidelity and network overhead characteristics.

If a client communicates the cr associated with a data item to the source of that data item, we could even make the server push the changes to the client according to the specified cr. We are currently developing an algorithm that adjusts the crs in a fashion similar to that of pull based approaches, but in this case, the adjusted cr is communicated to the server of the data. It is even possible to envisage a non-symmetric window of temporal coherency as opposed to the symmetric window implied by the cr based constraint.

In summary, we achieve our objective by considering the stocks in a portfolio as a semantic unit, in the sense that it reduces the polling frequency of all the stock of the portfolio if the stock prices are changing such that there is very little chance of the portfolio being satisfied. Whereas developed in the context of financial information such as stock prices, our approach can also be used to answer similar queries on any kind of dynamic data (e.g., traffic conditions, sensor data).

IV. Towards the Continuous Querying of Arbitrary Time-varying Data

Finally, in this section, we discuss a proposal for a full fledged system to handle continuous queries over a virtual database containing data from arbitrary data sources. For example, a patient status monitoring system, unlike a stock portfolio monitoring system, will have to cater to diverse data such as blood pressure, body temperature, etc. each measured in its own unit and not combinable in the way a portfolio was built from individual stock prices wherein all contributors were measured using the same unit. Since such a system must "think" on the fly and adapt its decision making, we have named it DeepThought, which can be configured as an appendage to a proxy on the Web.

The main features (and design goals) of the DeepThought system are:

- **Tailored**. The system is specifically designed for dynamic data. All operations are *memory based* and the system tries to minimize the time taken for processing at the proxy. Also, the delivery of data from the server to the proxy will be governed according to the query conditions with minimum possible delays. The DeepThought query language is also specifically tailored for dynamic data.
- **Distributed**. To ensure scalability and availability on the Internet scale.
- **Flexible**. Since there are various algorithms for data delivery between the server and proxy, the system chooses the best algorithm for a particular query/connection combination. DeepThought can use any of Pull, PoP, PaP or PoPoPaP for data retrieval.

Figure 1 shows the complete structure of the DeepThought system. The main interface of the DeepThought system to the user side is via the CQ Server Interface (CQSI), which may be a separate machine or a router machine. The user initially connects with the CQSI and asks for service. If the CQSI determines that the user can be serviced (depending on load at the proxy), then it responds with the query interface, otherwise a negative acknowledgement is sent and the user may attempt again later. The user, after receiving the query interface, enters the query (either manually or using some tool) and this query is sent to CQSI as a text string. We are not including the syntax description of DTQL, the DeepThought Query language for want of space, but it is a language specifically tailored for dynamic data. The following actions are taken after this,

- When a CQ is issued, the CQ Parser (CQP) breaks it up into A, T and R,
- The CQ Installer installs the CQ
- A thread/process called CQ Thread Monitor (CQTM) is started for the further processing of the CQ. This is preferably done on another machine in order to achieve *load sharing*. The CQTM does the job of evaluating the action part (A) of the CQ and coordinate the various tasks between components.
- the CQSI returns the CQ identifier to client as well as the chosen CQTM machine identity. The client now connects to the chosen CQTM and downloads an applet to obtain data from the machine.

The Source Monitor (SM) is the component which actually brings the data from the data source. There is only one SM for each data item in the whole system. This saves a lot of effort and bandwidth. SM uses an existing algorithm module to deliver data to TH. The algorithm modules may be for algorithms like Pull, Push, PaP, PoP [9], [4] etc. The algorithm module is pluggable, and SM provides a uniform interface for registry of the module, negotiation of the parameters and data delivery. The SM chooses the best algorithm depending on various parameters like server capabilities, volume of data through the connection, load on the SM etc. So we can use the most suitable algorithm for data retrieval. The SM is a separate process which may reside on a different machine (thus improving the bandwidth utilization).

Once the CQ has been parsed and installed, the actual execution of the query starts. The trigger, T, is installed by a separate entity called the Trigger Handler, following which:

- The CQTM passes the trigger part T to TH, together with some meta information and the list of items to be monitored.
- The TH installs T and gets it ready for repetitive execution.
- For each of the data items TH tries to determine if there is already a Source Monitor for a data item. If there is, then it is used or a new SM must be started. The SM pushes a new data value for a data item to the TH, whenever it is available.
- When new data value arrives, the TH evaluates the trigger T and the termination condition R, and returns the resulting data to the CQTM which can then take action A on it.

When the SM uses Pull-based algorithms, TH conveys polling intervals to the SM for each data item. For Push-based algorithms, this is not required. This must be done by TH since the final goal of the trigger is known only to TH through the trigger condition T. Depending on T, TH decides polling frequencies for each of the data items in T. The process of determining polling frequencies will be a variation of the one developed for portfolio queries. We are in the process of developing such a system and experience with its functioning will be reported in a future article.

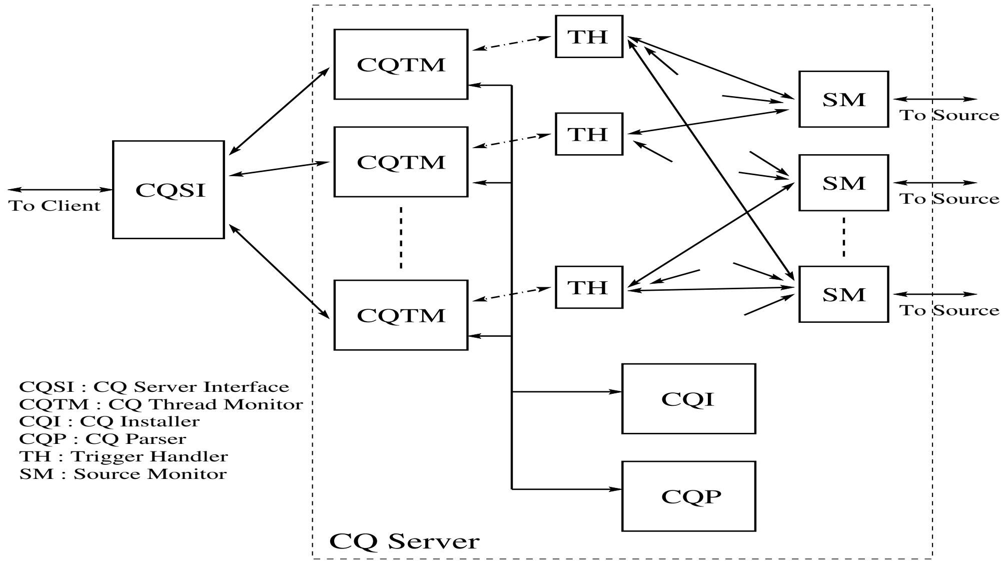

Fig. 1. The DeepThought System design

V. Challenges in Handling Dynamic Data on the Web

Pull and push approaches are typically used to retrieve data from the Web. Due to the specific nature of dynamic data, none of these approaches is a perfect fit: While pull suffers from lack of fidelity, push suffers from lack of resiliency and scalability. Our work demonstrates the need for various combinations of pull and the push approaches. Most notably, we show how fidelity, resiliency and scalability of the system improves by using PaP and PoP. This science of combination of push and pull can be put to innovative use in continual queries made on dynamic data. Our continual query system is probably the first one to deal specifically with dynamic data.

But there are many challenges in catering to the many complex types of queries that are likely to arise. For example, the queries that have been seen so far only consider positive threshold levels. In the general case, there could be portfolio queries wherein the proxy could be asked to monitor the difference between two data items. Such queries can be called *differential queries*. Examples of such kind of queries could be *" Tell me when the difference between the price of Cisco stock and Intel stock becomes greater than \$ 20"* [10] or *" Tell me when the blood pressure of a patient goes up by 10 units and his pulse crosses 100"*. Another type of query is what could be called a *gradient query*, whose result is based on the trend in data values. The problem in using the standard methods for differential queries is that constraints cannot be assigned directly to the constituent stocks. One way to handle such queries is to assign individual thresholds to each of the stocks. Further, the slope of the price change of the stocks also needs to be monitored and based on the direction of this change relative to the other changes, the thresholds need to be adjusted. With portfolios constituting of many stocks, the problem can be quite computationally expensive and efficient mechanisms need to be devised to solve this.

An astute reader would have noted the feedback driven nature of many of the decisions made by algorithms such as PoP, PaP, as well as their continual query counterparts. One of the challenges is the adjustment of the parameters used by these algorithms in order to achieve the feedback. The parameters should themselves adapt taking into account the current input and system dynamics. Currently we are investigating the usefulness of utilizing control theoretic principles to address the tuning of the parameters that drive such feedback. The talk will illustrate the results of this line of research.

There are several topics which deserve further attention in the context of time sensitive data dissemination:

- In a proxy/cache based environment, a proxy acts as a client to the provider of its data and as a server to its clients downstream. Each such server-client pair needs to decide the dissemination approach to use between them. Making these decisions in a decentralized fashion while globally achieving temporal coherency with low overheads and high resiliency is an interesting problem.
- We need to develop algorithms for disseminating time sensitive content through a content distribution network (CDN). Currently deployed CDNs are mostly concerned with storing static content, such as images, close to the potential clients of such data. It is perhaps even more important to use such an infrastructure for time varying data because of the potential for CDNs (and the underlying application level multicasting) to reduce communication and computation overheads.
- Support for disseminating time sensitive data in mobile and broadcast environments is another challenge. Given the wireless bandwidth restrictions in such environments, it is important to cater to user needs without explicit requests from users. This calls for push based approaches, but the challenge lies in obtaining user request profiles on a dynamic basis. An extension of the approach suggested in [**?**] might prove useful: a fraction of the bandwidth is allocated to each mode, periodic broadcast and on-demand broadcast. (This can be viewed as the broadcast analog of PoP.) The fraction allocated to each mode is varied with time depending on the access frequency distribution of data. To achieve this, the server needs some feedback from the clients about their access patterns.

Acknowledgements

This work was supported by Tata Consultancy Services through their funding for the Center for Intelligent Internet Research at IIT Bombay. Additional support was provided by the National Science Foundation Grant CCR-0098060. I am grateful to Mukund Agarwal, Manish Bhide, and Faizaan Marolia for their comments on an earlier version of this paper and to Prashant Shenoy for his collaboration on several aspects of the work reported here.

REFERENCES

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

[2] M. Bhide, K. Ramamritham and P. Shenoy, Efficiently Maintaining Stock Portfoilis up-to-date on the Web, (to appear in) *IEEE Research Issues in Data Engineering (RIDE'02) workshop*, March 2001.

[3] J. Chen, D. Dewitt, F. Tian and Y. Wang, NiagraCQ: A Scalable Continuous Query System for Internet Databases. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*.

[4] 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.

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

[6] J. Fernandez and K. Ramamritham, Adaptive Dissemination of Data in Asymmetric Communication Environments, ACM Mobile Networks & Applications Journal (to appear).

[7] B. Krishnamurthy and C. Wills: Piggyback Server Invalidation for Proxy Cache Coherency. Proc. World Wide Web Conference, April 1998.

[8] 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.*

[9] R. Srinivasan, C. Liang and K. Ramamritham, Maintaining Temporal Coherency of Virtual Data Warehouses, *The 19th IEEE Real-Time Systems Symposium (RTSS98), Madrid, Spain, December 2-4, 1998*.

[10] B. Urgaonkar, A. Ninan, M. Raunak, P. Shenoy and K. Ramamritham, Maintaining Mutual Consistency for Cached Web Objects *International Conference on Distributed Computing Systems , April 2001.*

[11] M. J. Franklin and S. B. Zdonik: "Data In Your Face": Push Technology in Perspective. SIGMOD Conference, Seattle, Washington, May-June 1998.