Dissemination of Dynamic Data

Pavan Deolasee Amol Katkar Ankur Panchbudhe
Krithi Ramamritham Prashant Shenoy

Department of Computer Science and Engineering.
Indian Institute of Technology Bombay
Mumbai, India 400076
{*pavan,amolk,ankurp,krithi*}@cse.iitb.ernet.in

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
{*krithi,shenoy*}@cs.umass.edu

Abstract—**An important issue in the dissemination of time-varying web data such as sports scores and stock prices is the maintenance of** *temporal coherency*. **In the case of servers adhering to the HTTP protocol, clients need to frequently** *pull* **the data based on the dynamics of the data and a user's coherency requirements. In contrast, servers that possess** *push* **capability maintain state information pertaining to clients and push only those changes that are of interest to a user. These two canonical techniques have complementary properties with respect to the level of temporal coherency maintained, communication overheads, state space overheads, and loss of coherency due to (server) failures. In this demonstration, we show how to combine push- and pull-based techniques to achieve the best features of both approaches. Our combined technique tailors the dissemination of data from servers to clients based on (i) the capabilities and load at servers and proxies, and (ii) clients' coherency requirements. By using our** *continual query* **system, we will show how diverse requirements of temporal coherency, resiliency and scalability can be met using our techniques.**

I. Introduction

Recent studies have shown that an increasing fraction of the data on the world wide web is time-varying (i.e., changes frequently). Examples of such data include sports information, news, and financial information such as stock prices. An important issue in the dissemination of this data is the maintainence of temporal coherency. Web proxy caches that are deployed to improve user response times must track such dynamically changing data so as to provide users with temporally coherent information. 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 data items such as stock prices. A proxy can exploit user-specified coherency requirements by fetching and disseminating only those changes that are of interest and ignoring intermediate changes. For instance, a user who is interested in changes of more than a dollar for a particular stock price need not be notified of smaller intermediate changes. We demonstrate a proxy based application tailored specifically for delivering dynamic data.

II. Major approaches for retrieving dynamic data

Consider a proxy that caches several time-varying data items. To maintain coherency of the cached data, each cached item must be periodically refreshed with the copy at the server. We assume that a user specifies a temporal coherency requirement tcr for 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. As shown in Figure 1, let $S(t)$, $C(t)$ and $U(t)$ denote the value of the data item at the server, cache and the user, respectively. Then, to maintain temporal coherency (tc) we should have

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

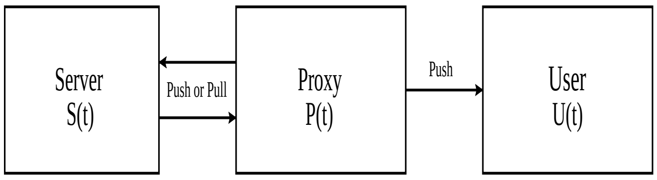

Fig. 1. The Problem of Temporal Coherency

A. The Pull Approach

To achieve temporal coherency (tc) using a pull-based approach, a proxy can compute a *Time To Refresh (TTR)* attribute with each cached data item. The *TTR* denotes the next time at which the proxy should poll the server so as to refresh the data item if it has changed in the interim. A proxy can compute the *TTR* values based on the rate of change of the data and the user's coherency requirements. Rapidly changing data items and/or stringent coherency requirements result in a smaller TTR, whereas infrequent changes or less stringent coherency requirement require less frequent polls to the server, and hence, a larger *TTR*. Observe that a proxy need not pull every single change to the data item, only those changes that are of interest to the user need to be pulled from the server (and the TTR is computed accordingly).

TABLE I
BEHAVIORAL CHARACTERISTICS OF DATA DISSEMINATION ALGORITHMS

Algorithm	**Resiliency**	**Temporal Coherency** (fidelity)	**Overheads (Scalability)**		
			Communication	*Computation*	*State Space*
Push	Low	High	Low	High	High
Pull	High	Low (for small tcrs) High (for large tcrs)	High	Low	Low

Clearly, the success of the pull-based technique hinges on the accurate estimation of the TTR value. A discussion on suitable techniques for determining TTR appears in [5]. We used the adaptive TTR approach mentioned therein.

B. *The Push Approach*

In a push-based approach, the proxy 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 proxy; only those changes that are of interest to the user (based on the tcr) are actually pushed. Formally, if D_k was the last value that was pushed to the proxy, then the current value D_l is pushed if and only if $|D_l - D_k| \geq c$, $0 \leq k \leq l$. To achieve this objective, the server needs to maintain state information consisting of a list of proxies interested in each data item, the tcr of each proxy and the last update sent to each proxy.

The characteristics of the approaches are summarized in table I

III. COMBINATIONS OF PUSH AND PULL

It is clear from table I that both the approaches mentioned have complementary properties in terms of fidelity achieved, scalability and resiliency. To achieve the best characteristics of both the approaches, it is essential to combine them in some way. While combinations have been suggested in the past [1], we have developed, implemented and evaluated these approaches. In fact, our approaches not only achieve high scalability, they also deal with resiliency and fidelity improvement. We now present three approaches to combine push and pull seamlessly so as to get the best of both methods to obtain these properties.

A. *The Push and Pull* (PaP) *algorithm*

Suppose a client registers with a server and intimates its coherency requirement tcr. Assume that the client pulls data from the server using an algorithm, say A (e.g., *Adaptive TTR*), to decide its TTR values. After initial synchronization, server also runs algorithm A. Under this scenario, the server is aware of when the client will be pulling next. With this, whenever server sees that the client must be notified of a new data value, the server pushes the data value to the proxy if and only if it determines that the client will take time to poll next. The state maintained by this algorithm is a soft state in the sense that even if push connection is lost or the clients' state is lost due to server failure, the client will continue to be served at-least as well as under A. Thus, compared to a Push-based server, this strategy provides for graceful degradation.

For the advantages of this technique to accrue, the server need not run the full-fledged TTR algorithm. A good approximation to computing the client's next TTR will suffice. For example, the server can compute the difference between the times of the last two pulls ($diff$) and assume that the next pull will occur after a similar $delay$, at $t_{predict}$. Suppose $T(i)$ is the time of the most recent value. The server computes $t_{predict}$, the next predicted pulling time as follows:
- let $diff = T(i) - T(i-1)$
- server predicts the next client polling time as $t_{predict} = T(i) + diff$.

If a new data value becomes available at the server before $t_{predict}$ and it needs to be sent to the client to meet the client's tcr, the server pushes the new data value to the client.

In practice, the server should allow the client to pull data if the changes of interest to the client occur close to the client's expected pulling time. So, the server waits, for a duration of ϵ, a small quantity close to TTR_{min}, for the client to pull. If a client does not pull when server expects it to, the server extends the push duration by adding $(diff - \epsilon)$ to $t_{predict}$. It is obvious that if $\epsilon = 0$, *PaP* reduces to push approach; if ϵ is large then the approach works similar to a pull approach. Thus, the value of ϵ can be varied so that the number of pulls and pushes is balanced properly. ϵ is hence one of the factors which decides the tc properties of the PaP algorithm as well as the number of messages sent over the network.

B. *The Push or Pull* (PoP) *Algorithm*

While PaP is good for achieving resiliency and temporal coherency, it is poor in terms of scalability. We now present PoP which dynamically chooses between push or pull in order that more clients can be served. PoP is based on the premise that at any given time a server can categorize its clients either as push clients or pull clients and this categorization can change with system dynamics. This categorization is possible since the server knows the parameters like the number of connections it can handle at a time and can determine the resources it has to devote to each mode (Push/Pull) of data dissemination so as to satisfy its current clients. The basic ideas behind this approach are:

• allow failures at a server to be detected early so that, if possible, clients can switch to pulls, and thereby achieve graceful degradation to such failures. To achieve this, servers are designed to push data value to their push clients when one of two conditions is met:

1. The data value at the server differs from the previously forwarded value by tcr or more.
2. A certain period of time TTR_{limit} has passed since the last change was forwarded to the clients.

The first condition ensures that the client is never out of sync with the values at the server by an amount exceeding the tcr of the client. The second condition assures the client after passage of every TTR_{limit} interval that (a) the server is still up and (b) the state of the client with the server is not lost. This makes the approach resilient. In case of the state of the client being lost or the connection being closed because of network errors, the client will come to know of the problem after TTR_{limit} time interval, after which the client can either request the server to reinstate the state or start pulling the data itself. This ensures that in the worst case, the time for which the client remains out of sync with the server never exceeds TTR_{limit}.

• In this approach, the server can be designed to provide push service as the default to all the clients provided it has sufficient resources.

• When a resource constrained situation arises (upon the registration of a new client or network bandwidth changes) some of the push-based clients are converted to become pull-based clients based on the criteria that we had determined earlier.

The state diagram for achieving this adaptation is shown in Figure 2.

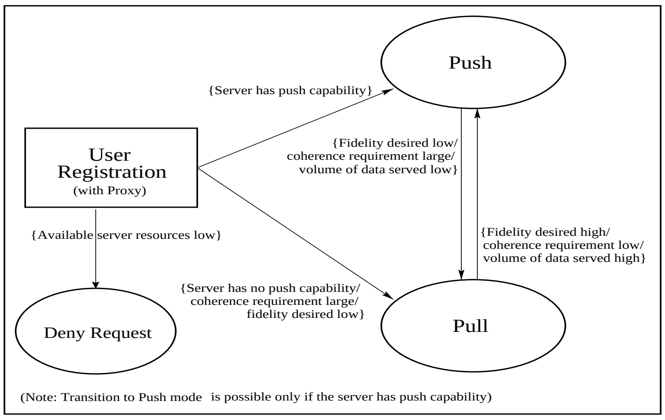
Push
{Server has push capability}
User Registration (with Proxy)
{Fidelity desired low/ coherence requirement large/ volume of data served low}
{Available server resources low}
{Fidelity desired high/ coherence requirement low/ volume of data served high}
{Server has no push capability/ coherence requirement large/ fidelity desired low}
Deny Request
Pull
(Note: Transition to Push mode is possible only if the server has push capability)

Fig. 2. PoP: Choosing between Push and Pull

C. *Combining PaP and PoP:* PoPoPaP

As mentioned before, *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*.

IV. *DeepThought:* THE CONTINUAL QUERY SYSTEM

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 the system to give the user the freedom to ask any query on the database. Continual queries allow users to obtain new results from database without having to issue the same query again and again. CQs are especially useful in an environment like the Internet, 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.

A. *Features of DeepThought*

The DeepThought system is:

• **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 is governed according by the query conditions. The DeepThought query language is 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.

B. *Comparison with existing CQ systems*

The basic idea of DeepThought is very similar to Conquer [4]. However, Conquer is tailored more for heterogenous data than for real time data. It uses disk-based database as backend and implements the system as an extension of this database. The query language is an extension of SQL and hence specifying queries unique to real time data is difficult.

NiagraCQ [2] is another CQ system for the Internet. The focus of this system is on scalability for the Internet. Similar queries or parts of queries are grouped logically so that these parts need not be re-evaluated again and again. Since most of the queries on the Internet are somewhat similar, this process certainly proves beneficial. NiagraCQ again uses disk-based databases. DeepThought provides scalability by distribution of effort. However, the dynamic nature of data makes identifying logically equivalent parts difficult. As against this, DeepThought extracts unique data items from various queries so that there is only one thread serving a single data item from the servers.

Also, all the existing CQ systems initiate actions when the database is updated by external sources. DeepThought makes an active contribution towards predicting when the system should look for changed data values.

C. The Architecture in short

Figure 3 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 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 comes, 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 [5], [3] 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 situation).

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

- 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 is 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,

D. Calculating Polling Frequencies

When SM uses Pull-based algorithms, TH conveys polling intervals to the SM for each data item. This is 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. Following is the process of determining polling frequencies:

- For each T, the TH creates a function $F(x_1, x_2, \dots x_n)$ which is a function of all the data item values (x_is) which the TH needs to monitor in order to check if T is true.
- Let x_{i1} and x_{i2} be two consecutive values obtained from the data source with a time duration of δt between the two data values. Since we have a trace of all x_is during this duration, we calculate the value of function F at the two instants of time, assuming change only in x_i. That is, we find

$$\delta F_i = F(x_{11}, x_{21} \dots x_{i2} \dots x_{n1}) - F(x_{11}, x_{21} \dots x_{i1} \dots x_{n1})$$

Like this, we find the effect of unit change in each of the variables x_i on the function in unit time.

- We determine the *polling frequency* for a particular data item by having it proportional to δF_i, for each of the data items x_i.
- Since there are multiple CQs interested in the same data item, we use the highest polling frequency of all of them as the frequency of polling for the particular data item.

Whenever a new data value comes, it changes the function F. But, since each of the data items are independent of each other, they may change F independently at different instants. Whenever one data item x_i changes F, all x_js ($i \neq j$) are notified of this change. Also, the change that x_i makes in F is not independent of x_js. So, suppose F_k was the value of F previously. When the new value of x_i

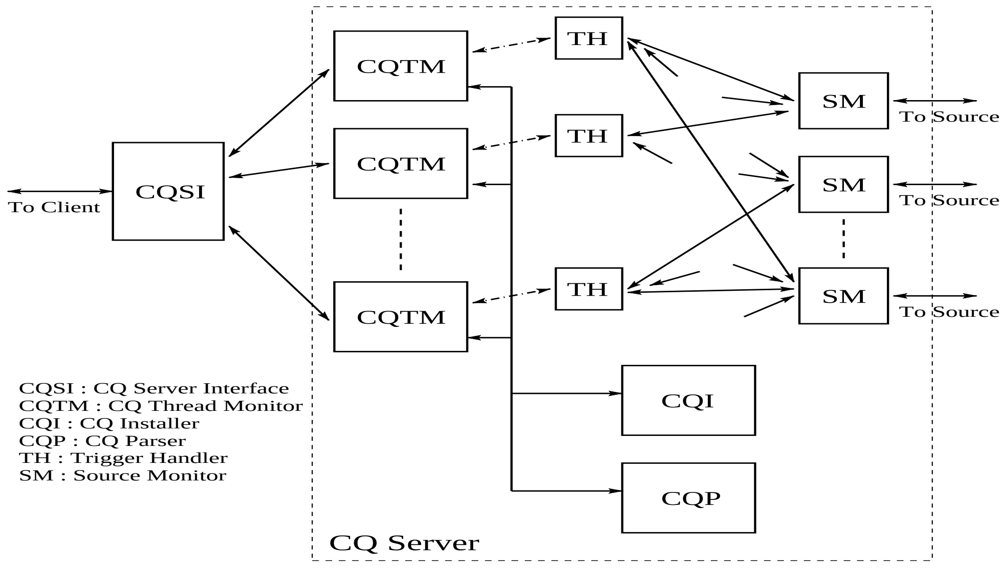

Fig. 3. The DeepThought System design

comes, we reevaluate F with the new value of x_i and old values of x_js. The new value of F becomes F_{k+1}. But since the polling frequencies of all x_js were decided on the basis of the value F_k and the value has now become F_{k+1}, their polling frequencies are readjusted. As explained above, we set the polling frequency of a data item according to the change it makes in F in unit time i.e. δF_i .

V. Demonstration

The major contributions of our research work are:

• Three new algorithms *PaP*, *PoP* and their combination *PoPoPaP* for disseminating dynamic data on the Internet. They achieve resiliency, scalability and temporal coherency.

• A totally new continual query system catering specifically to dynamic data

We will be demonstrating the working of the entire system in specific reference to the stock market scenario.

• The users will be able to enter queries such as:

```
show v1=(100*MSFT+200*INTL+300*IBM)
for stocks when change(v1) > 500$
till v1 > 20000$;
```

on remote servers. The specific instance mentioned above is a portfolio query.

• Get stock related Web data.

• Choose between Pull, PaP, PoP and PoPoPaP and determine the efficacy of our approaches.

VI. Conclusion

The pull and push approaches are used to retrieve data from the Web. Due to the specific nature of the data that we are dealing with, none of these approaches can in itself retrieve data on the web scale. While pull suffers from lack of fidelity, push suffers from lack of resiliency and scalability. This demonstration demonstrates the efficacy of various combinations of pull and the push approach. 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. We have a continual query system which is

• Uses only main memory databases so that delays associated with disk based databases which become significant when dealing with real time data can be eliminated.

• Has a query language specifically tailored to deal with dynamic data.

• First system to do computation dissemination

• Uses a novel method for retrieving data from the web.

• Is fully distributed.

We will be demonstrating the CQ system and using it as a showcase, demonstrate the efficacy of our algorithms.

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] 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.*

[3] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham and P. Shenoy, Dissemination of Dynamic Data on the Internet. *International Workshop on Databases in Networked Information Systems, University of Aizu, JAPAN December 4-6, 2000.*

[4] 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.*

[5] 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.*