QoS Management in Web-based Real-Time Data Services *

Sang H. Son and Kyoung-Don Kang
Department of Computer Science
University of Virginia
{son, kk7v}@cs.virginia.edu

Abstract

The demand for real-time data services has been increasing recently. Many e-commerce applications and web-based information services are becoming very sophisticated in their data needs. They span the spectrum from low level status data, e.g., stock prices, to high level aggregated data, e.g., recommended selling/buying point. In these applications, it is desired to process user requests within their deadlines using fresh data, which reflect the current market status. Current web-based data services are poor at processing user requests in a timely manner using fresh data. To address this problem, we present a framework for guaranteed real-time data services in unpredictable environments. We also introduce a possible application of our approach in the distributed environment.

1 Introduction

The demand for real-time information services has been increasing recently. Many e-commerce applications and web-based information services are becoming very sophisticated in their data needs. They span the spectrum from low level status data, e.g., current stock prices, to high level aggregate data, e.g., recommended buying/selling point. In e-commerce applications, it is important to process transactions within their deadlines using fresh (temporally consistent) data reflecting the current market status. In this way, merchandise such as stock items can be traded in a timely manner before the current market conditions change. In this paper, we consider that real-time data services are provided if these timing and data freshness constraints can be met.

It is challenging to provide real-time data services because of data-dependent resource requirements and highly uncertain workloads in web/e-commerce applications. For example, transactions in stock trading may read different sets of stock prices and perform different operations based on the current market status. Furthermore, the number of service requests can drastically vary in an open environment such as World Wide Web. Consequently, developing real-time data services should involve techniques for managing overloads and unexpected failures to support the performance requirements expressed in terms of deadline miss ratio and data freshness. In this paper, we present a novel QoS management scheme for real-time data services, which provide certain guarantees on deadline miss ratio and data freshness even in the presence of unpredictable workloads and data access patterns.

Unfortunately, timeliness and data freshness requirements can conflict with each other. The deadline miss ratio of user transactions, e.g., trade requests in e-commerce, can be decreased by giving a higher priority to user requests. In contrast, better freshness can be achieved by favoring updates [1]. Effective balancing between the user requests and update workload is the key to meet both the specified deadline miss ratio and data freshness constraints. For this purpose, we present a *dynamic balancing* scheme to balance the user transaction and update workloads.

Performance control issues in unpredictable environments have been well studied in the feedback control theory. We propose to apply a feedback control approach to provide robustness against unpredictable workloads. In a feedback control system, target performance can be achieved by dynamically adjusting the system behavior based on the error measured in the feedback loop. A preliminary performance study indicates that our QoS-sensitive approach can achieve a significant performance improvement, in terms of deadline miss ratio and data freshness, compared to several baseline approaches [5].

The rest of this paper is organized as follows. In Section 2, real-time data service semantics are discussed. Section 3 describes performance metrics and discusses QoS specification issues. Section 4 presents a QoS management scheme to provide deadline miss ratio and data freshness guarantees. Section 5 discusses data replication issues in the distributed environment to improve timeliness and service availability. Related work is presented in Section 6

* Supported, in part, by NSF grant EIA-9900895 and CCR-0098269.

and Section 7 concludes the paper.

2 Real-Time Data and Transactions

In this section we give an overview of real-time data and transactions.

2.1 Real-Time Database Model

Our target applications are firm real-time data services, in which tardy transactions − transactions that have missed their deadlines − add no value to the system, therefore, are aborted. In this paper, we consider a main memory database model. Main memory databases have been increasingly applied to real-time data management such as online auction/stock trading, e-commerce and voice/data networking [1, 3, 19]. In our main memory database model, the CPU is the system resource of main consideration.

2.2 Temporal Data Management

Data can be classified into two classes: non-temporal and temporal. A non-temporal data object does not become outdated due to the passage of time, e.g., PIN numbers. In contrast, a temporal data object may change continuously to reflect the real world state, e.g., current temperature or stock prices. Each temporal data object has a timestamp indicating the latest observation of the real-world state.

Temporal consistency was defined using validity intervals in a real-time database to address the consistency issue between the real world state and the reflected value in the database. A temporal data object X is considered fresh (temporally consistent) if $(current\ \ time - timestamp(X) \leq avi(X))$, where $avi(X)$ is the absolute validity interval of X. Thus, absolute validity interval is the length of the time a temporal data object remains fresh [16].

Temporal data can be further classified into *base data* and *derived data*. Base data objects import the view of the outside environment. A derived data object can be derived from possibly multiple base/derived data. For example, a composite index can be derived from various stock prices. A base data item is directly associated with an absolute validity interval. A derived data object is associated with absolute and relative validity intervals [16].

2.3 Transaction Types

Currently, our real-time database model includes two types of transactions:

• *Update Transactions*: In a real-time database, base data objects are updated periodically to reflect the current status of the real-world environment, e.g., current stock price. Update transactions are write-only transactions specially designed for this purpose.

• *User Transactions*: User transactions arrive aperiodically. User transactions do not write any temporal data object, but they can read/write non-temporal data. A user transaction can perform arithmetic/logical operations based on a set of temporal/non-temporal data. Execution times and data access patterns of user transactions can be time-varying as discussed before.

We assume that each user transaction has a deadline. The deadline of an update transaction is set to the update period. A tardy transaction is aborted. For scheduling transactions, we apply earliest deadline first (EDF) algorithm [8] combined with our adaptive update scheduling policy.

Admission control can be applied to user transactions. A newly arrived user transaction is admitted to the system if the requested CPU utilization is currently available. The current utilization can be examined by aggregating the utilization estimates of the previously admitted transactions.

3 QoS Specification

In this section, we describe the performance metrics considered in our approach. QoS specification issues are also discussed using the performance metrics.

3.1 Average Performance Metrics

• *Deadline Miss Ratio of User Transactions (MR)*: In a QoS specification, a DBA can specify the target deadline miss ratio that can be tolerated for a specific real-time application.

• *Data Freshness*: We categorize data freshness into *database freshness* and *perceived freshness (PF)*. Database freshness is the ratio of fresh data to the entire temporal data in a database. Perceived freshness is the ratio of fresh data accessed to the total data accessed by timely transactions − transactions which finish within their deadlines. To measure the current perceived freshness, we exclusively consider timely transactions. This is because tardy transactions, which missed their deadlines, add no value to the system. Note that only the perceived freshness can be specified in the QoS requirement. Our QoS-sensitive approach provides the perceived freshness guarantee while managing the database freshness internally (hidden to the users).

3.2 Transient Performance Metrics

Long-term performance metrics such as average miss ratio are not sufficient for performance specification of dynamic systems such as web information systems, in which the system workload and performance can be widely time-varying. For this reason, we use transient performance metrics, adopted from control theory, as performance metrics for real-time data services [11]:

• *Overshoot* (M_p) is the worst-case system performance in the transient system state. In this paper, it is considered the highest miss ratio over the miss ratio threshold in the transient state. In general, a high transient miss ratio may imply a loss of profit in e-commerce.

• *Settling time* (t_s) is the time for the transient overshoot to decay and reach the steady state performance.

For example, a DBA can require to limit MR below 5% while achieving at least 98% PF. The DBA can also require to limit M_p and t_s below 6.5% and 45sec, respectively. By controlling both average and transient performance, the performance of highly dynamic systems, e.g., web information systems, can be managed in a consistent manner.

3.3 Other Performance Metrics

We also consider two additional performance metrics:

• *Differentiated Timeliness:* In QoS requirements, relative response time between service classes can be specified. For example, relative response time can be specified as 1:2 between premium and basic classes. By allowing and enforcing the explicit specification, we can guarantee the distance between service levels of different classes. Therefore, the performance can be differentiated in proportion to the importance of service classes. It will be interesting to find out if relative service differentiation can be actually achieved, while satisfying both data freshness and transaction timeliness, given unpredictable workload.

• *Freshness of Derived Data:* To maintain the freshness, a derived data object has to be recomputed as the related base data changes. A recomputation of derived data can be relatively expensive compared to a base data update. It can include arithmetic and/or logical computations from multiple base data, which might be updated frequently. Therefore, it could be computationally very expensive if a derived data object is recomputed on each base data change.

Due to space limitations, we refer readers to [4] and [17] for more details regarding timeliness differentiation and derived data management, respectively.

4 Feedback Control for QoS Management

In this section, we first present the feedback control approach for QoS management in real-time data services, and then introduce a cost-benefit model to quantify the utility of an update for a specific temporal data object. Using this model, we propose a database QoS adaptation technique for managing data freshness. In general, the update policy is not dynamically changed in databases. In contrast, our QoS adaptation technique dynamically adjusts the update policy based on the current workload and system behavior to provide guaranteed real-time data services.

4.1 Feedback Control

It is a challenging research problem to provide certain guarantees on transaction timeliness and data freshness, given aperiodic transaction arrivals and unpredictable workloads. An advantage of the feedback control scheduling is its use of feedback control theory, rather than ad hoc solutions, as a scientific underpinning. The control theory based design methodology allows to satisfy the steady state as well as transient performance specifications.

We have been working on developing a framework for QoS management of real-time data services, based on feedback control theory [5]. Using this framework, a designer can systematically design an adaptive QoS manager to satisfy the performance specifications with established analytical methods. This approach contrasts to existing ad hoc methods that depends on design/testing/tuning iterations. The outline of our framework is as follows.

• The designer specifies the desired behavior using steady state as well as transient state performance metrics. This step maps the existing metrics of real-time data services to the dynamic responses of control systems.

• The designer establishes a dynamic model of the real-time data services for performance control. The model describes the relationship between the control input and the performance of the system with differential/difference equations. Modeling is important because it provides the basis for the analytical design of the QoS controller. To establish a dynamic model, system identification method [15] can be used to estimate the system behavior using profiling experiments. Our earlier work on system identification of web servers using profiling shows that such an approach could work well for software systems [10].

• Based on the system model and the performance specifications from previous steps, the designer applies existing mathematical techniques (e.g., the Root Locus method or state-space design) of feedback control theory to design the feedback controller with analytic guarantees on the desired transient as well as steady state behavior.

For real-time data services, we apply a feedback control real-time scheduling policy, called FC-UM [12], to manage the miss ratio without under-utilizing the CPU in the presence of unpredictable workloads. As shown in Figure 1, FC-UM employs two control loops, one for miss ratio and one for CPU utilization management. FC-UM achieves the stability by integrating two control loops. The intuition behind this combination lies in the fact that the saturation conditions of the two control loops are mutually exclusive. A utilization controller is saturated at 100% utilization, while a miss ratio controller can be saturated when the real-time system is underutilized (0 miss ratio as a result). Each control loop generates a control signal to achieve the target utilization or miss ratio based on the current performance error, which is the difference between the target miss ratio (utilization) and the currently measured miss ratio (utilization). Each controller computes the control signal, called requested CPU utilization adjustment ΔU, to achieve the target miss ratio (or target utilization).

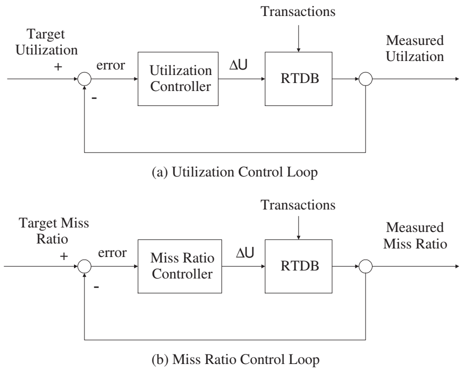


Figure 1. Miss Ratio/Utilization Controllers

We have selected FC-UM to manage QoS for real-time data services, since it can meet our basic requirements for real-time scheduling: miss ratio guarantees while not underutilizing the CPU given a mix of periodic and aperiodic workloads. Another advantage of FC-UM is that it works well even without a precise workload model. We adapted and tuned the feedback controllers. For more details, refer to [17].

In our model, data freshness is managed by an actuator (i.e., database QoS manager) in the real-time database. We do not consider designing a separate feedback controller for freshness management since timeliness and freshness can pose conflicting requirements, leading to a potentially unstable feedback control system.

The interactions between FC-UM and our QoS management scheme are described in Figure 2. When ΔU is negative (indicating a violation of the specified miss ratio), QoS in terms of database freshness can be degraded to improve the miss ratio if the current perceived freshness is above the target and the degradation bound is not reached yet. (More discussions shall be given in the following sections.) When the perceived freshness is violated, the QoS will be upgraded as long as $\Delta U > 0$ and a certain upgrade bound is not exceeded. The corresponding QoS upgrade/degradation stops when either of the conditions is not satisfied. Without these conditions, a QoS degradation (upgrade) could adversely affect freshness (miss ratio) in the next sampling period, leading to possible oscillations between many deadline misses and data freshness violations.

In an extreme case, not only miss ratio, but also freshness constraints can be violated. In this case, incoming transactions will be rejected until a fraction of currently running transactions terminate and ΔU becomes positive as a result. The chance of an extreme case is reduced by enforcing the QoS adaptation conditions as described before. Furthermore, feedback control scheduling and admission control can prevent a severe overload.

1. Collect access statistics and compute AUR. (Section 4.2)
2. Monitor the miss ratio (MR), CPU utilization, and perceived freshness (PF).
3. At each sampling period, compute the miss ratio and utilization control signals. Set ΔU to the minimum of the two control signals for a smooth transition from one system state to another. Based on ΔU, perform one of the following alternative actions.
4. If $MR \leq MR_t$ (target miss ratio), i.e., $\Delta U \geq 0$, and $PF \geq PF_t$ (target perceived freshness), admit more transactions to prevent a potential underutilization.
5. If $MR > MR_t$, i.e., $\Delta U < 0$, and $PF \geq PF_t$, degrade the QoS. Increase ΔU by the utilization saved from a QoS degradation. Repeat until $\Delta U \geq 0$ or the degradation bound is reached. Inform the admission controller of the new ΔU.
6. If $PF < PF_t$ and $\Delta U > 0$, upgrade the QoS. Subtract the required utilization for an upgrade from ΔU. Repeat until $\Delta U \leq 0$ or a certain upgrade bound is reached. Inform the admission controller of the new ΔU.
7. If $MR > MR_t$ and $PF < PF_t$, do not admit any incoming transaction until ΔU becomes positive after some of the currently running transactions terminate.

Figure 2. QoS Management Scheme for Miss Ratio and Freshness Guarantees

4.2 A Cost-Benefit Model of Updates

In a real-time database, the temporal data update workload might be high. For example, in the NYSE trace the update stream can reach up to 696 updates/sec [7]. If updates receive higher priority, there may not be enough time left to finish transactions in time. In contrast, if the transactions are scheduled in a preferred manner, they may have to read stale data [1]. To balance the update and transaction workload efficiently, we need to estimate the update utility which captures the cost-benefit relation of temporal data updates. The cost is defined as the update frequency of a temporal data object. To consider the benefit, access frequency is measured for each data object. If a data object is accessed frequently, e.g., a popular stock price, an update of the object can produce a relatively high benefit. To quantify the cost-benefit relationship, we define the update utility, called Access Update Ratio (AUR), for a data object O_i as follows:

$$AUR[i] = \frac{\textit{Access Frequency}[i]}{\textit{Update Frequency}[i]} \quad (1)$$

The notion of AUR has several interesting features. It can be a guideline to decide a proper update policy for a certain data object. If AUR ≥ 1 for a data object, the benefit of the corresponding data update is worth the cost, since it is accessed at least as frequently as it is updated. Otherwise, it is not cost-effective. For simplicity, let us call a data object *hot* if AUR ≥ 1; otherwise, *cold*.

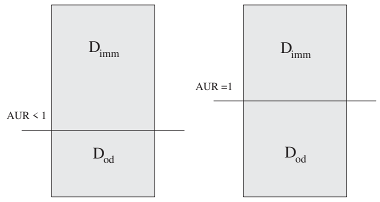
It is reasonable to update hot data in an aggressive manner. If a hot data object is out-of-date when accessed, potentially a multitude of transactions may miss their deadlines waiting for the update. For this reason, we immediately update hot data objects, e.g., hot stock prices, when their values change. Alternatively, it may not be necessary to immediately update cold data when overloaded. Only a few transactions may miss their deadlines waiting for the update. Under overload, the CPU utilization can be reduced by updating cold data on demand. In fact, cold data could always be updated on demand regardless of the current system load. However, that approach may increase the response time of the accessing transaction.

An immediate update receives a higher priority than user transactions and on-demand updates. Conceptually, there are separate scheduling queues for immediate updates and for user transactions/on-demand updates. Immediate updates in the high priority queue are scheduled before user transactions and on-demand updates in the low priority queue. In each queue, transactions are scheduled by EDF scheduling algorithm. As a result, the freshness of cold data can be relatively low compared to hot data, if a lazy update policy is applied to cold data.

Note that the notion of AUR does not depend on a specific access pattern or popularity model. It can be derived simply from the known update frequency and monitored access frequency. Therefore, it greatly simplifies our QoS model and makes the model robust against potential unpredictability in data access patterns.

4.3 Dynamic Adaptation of Updates

Figure 3 gives a pictorial example of our approach to dynamic update adaptation based on the cost-benefit model. In Figure 3, D represents the set of all temporal base data in the database, in which D_{imm} is the set of data updated immediately, and D_{od} stands for the set of data updated on demand. Since a data object is updated either immediately or on demand, $D = D_{imm} \bigcup D_{od}$ and $D_{imm} \bigcap D_{od} = \emptyset$. Initially, every data is updated immediately. As the load increases, a larger fraction of cold data objects are updated on demand. This is called *degradation*, since it may potentially degrade data freshness. The update policy degradation stops once it reaches a certain degradation bound, namely $AUR = 1$.



(a) Initial State (b) Moderately Loaded State (c) Overloaded State

Figure 3. Update Policy Adaptations

Several issues need to be addressed in this approach. First, access statistics need to be collected to compute $AUR[i]$ for each temporal data object O_i. On each access of O_i, the access frequency $AF[i]$ is incremented. However, the access frequency may have a large deviation from one sampling period to another. To smooth the potentially large deviation, we take a moving average at the k_{th} sampling instant:

$$SAF_k[i] = a \times SAF_{k-1}[i] + (1-a) \times AF_k[i] \quad (2)$$

where $0 \leq a \leq 1$. As the value of a gets closer to 0, only the recent access frequencies are considered to compute the moving average. In contrast, the wider horizon will be considered to compute the moving average as a get closer to 1.

Since the update frequency $UPDATES[i]$ in a sampling period is known for periodic updates of O_i, we can compute Access Update Ratio for O_i:

$$AUR[i] = \frac{SAF_k[i]}{UPDATES[i]} \quad (3)$$

To handle aperiodic updates, the update frequency can be monitored and smoothed in a similar way as the access frequency. For aperiodic updates the definition of data freshness may have to change, since there might be no explicit notion of validity interval related with aperiodic updates. In that case, a temporal data object can be considered stale upon the arrival of the corresponding update, which is not applied yet [2]. An important question is how to estimate the CPU utilization saved from each degradation. We need to find out the difference between the required CPU utilization for purely immediate updates and for adaptive updates. The number of saved updates due to the degradation is approximately:

$$N = UPDATES[i] - SAF[i] \tag{4}$$

Given the average CPU utilization per single update transaction, σU, the saved CPU utilization from the update policy degradation for O_i is approximately:

$$\delta U = N \times \sigma U \tag{5}$$

Average per update utilization, σU, can be either pre-profiled before the database service initiation, or measured at run time. Either approach may not introduce a considerable error, since each update transaction is known a priori and fixed in our real-time database model.

After the update policy degradation for a single data object, the new CPU utilization adjustment is:

$$\Delta U = \Delta U_{old} + \delta U \tag{6}$$

The degradation continues for the next data object in D_{imm} with the least AUR until $\Delta U \geq 0$ or the degradation bound is reached. AUR = 1 is the *update policy degradation bound*. Further degradation past the degradation bound is meaningless because the number of updates may not be reduced if hot data object is updated on demand, while accessing transactions may suffer unnecessary delay for update. An example of update policy degradations is shown in Figure 3. Initially, every update is performed immediately. The update policy is gradually degraded as the load increases. The degradation stops if no more adaptation is allowed either by the violation of perceived freshness or reaching the degradation bound.

In QoS upgrades, the update policy is switched back to the immediate policy for certain data objects to ensure that timely transactions access fresh data. An important issue in QoS upgrades is to avoid a potential miss ratio overshoot in the next sampling period, while improving the perceived freshness as needed. Given a target perceived freshness F_{target}, the current perceived freshness $F_{current}$ can be measured in a sampling period. If $F_{current}$ is less than F_{target}, some of these data should be updated immediately in the next sampling period. The freshness can be improved approximately in proportion to the number of data moved from D_{od} to D_{imm}. The key question is how to estimate the number of data (and which ones) to be updated immediately. More details of QoS upgrade/degradation are discussed in [5].

Due to the approximation, unpredictable workloads and access patterns, our QoS adaptation may not be precise. However, the target performance can be achieved by continuously adjusting the QoS level based on the performance error measured in the feedback loop. By balancing update and transaction workload efficiently, a target deadline miss ratio and perceived freshness can be achieved at the same time. It will be interesting to investigate the accuracy of the approximation of CPU utilization and perceived freshness, and their impact on the feedback controller. That could provide a quantitative measure of the effectiveness of the proposed QoS management approaches in terms of guaranteeing the required timeliness and data freshness.

5 Data Replication in a Distributed Environment

While a number of problems must still be addressed in a centralized environment for managing QoS, distributed systems issues are also critical for improving the performance of real-time data services. Although there are many issues in a distributed environment in general, in this section, we focus on data replication. Data replication has been used to improve the performance and availability in distributed applications. Performance in terms of transaction timeliness can be improved since replicated copies are available at the site where they are frequently accessed. Availability can be improved since the sites at which they are stored have independent failure modes. However, theses benefits come at the cost of overhead associated with update transactions that can be committed only after all sites containing replicated copies agreed to commit to enforce mutual consistency. This approach, called synchronous replication, has serious performance problems in practice [14]. Some commercially available distributed database systems such as Sybase and Oracle 7 support synchronous replication as well as lazy (asynchronous) replication, in which the updates of replicated data objects can be deferred. With lazy replication, a transaction can commit without waiting for all the replicated copies to be updated, relaxing the mutual consistency property. For lazy replication, data freshness is a key concern.

For web-based information services, replication of data/information using proxies and web caching has been popular. Although web caching and proxies addressed the scalability problem in providing desired performance for static contents, it introduced another problem of data freshness for dynamic contents. As the number of web pages that contain dynamic contents that are continuously changing is increasing rapidly, providing any guarantee for the freshness of data becomes important.

We consider the problem of maintaining acceptable freshness in lazy replication by extending our QoS management scheme. For each replicated temporal data object, there is a primary copy (denoted by capital letters) which is updated periodically. We call the site which contains the primary copy of a data object a master node (of that data object). Other copies are called secondary copies and the sites which contain secondary copies are called slave nodes. The set of replicated copies, consisting of the primary copy (e.g., R) and all secondary copies (r_1, r_2, ...), is called *replicated data*.

When an application starts to run at a site which does not have the temporal data object it needs, a secondary copy (r_i) is created and initialized by the current value of the primary copy (R). Secondary copies can be updated by either push or pull. We consider the pull model first, because it reduces the overhead at the master node and provides flexibility to each slave node to change its access pattern (i.e., a slave node does not need to inform the master node to make any change on its behalf). Each slave node pulls the value of the primary copy at the master node periodically, considering the local workload and data access statistics. The period of pull can be dynamically adapted by the QoS manager of the node using the current status information on timeliness, freshness, and utilization. Some of the cold data might be pulled on demand as discussed in the previous section.

We assume that when initially created, all the secondary copies are updated at the same rate of the primary copy. Each slave node keeps track of the access pattern for its secondary copies. Using Access Update Ratio (AUR) as discussed in Section 4.2, the slave node computes the candidates of its secondary copies for delayed update. When an overload occurs at a node, it can now adjust the CPU utilization in several ways. It can switch its update policy from immediate update to on-demand update for some of its primary copies. Alternatively, it can delay updates for some of its secondary copies. Another possibility is to consider both primary copies and secondary copies together. Instead of giving priority to one over the other, whichever copy that brings in higher access demand (or being accessed by high class/priority transactions) will be kept fresh as much as possible – either by updating immediately (primary copies) or pull the value of the primary copy at the initial rate.

To provide appropriate QoS over replicated data in distributed environment, there is another issue to deal with: *update scheduling problem*. Since secondary copies are being refreshed constantly, a separate update scheduler is responsible for scheduling the updates, using the portion of the CPU utilization allocated for refreshing secondary copies at the node. The primary objective of the update scheduler is to satisfy the specified QoS by achieving the maximum overall perceived freshness of replicated data. Since the cost of refreshing secondary copies could be different in general for each replicated data, the update scheduler should consider the frequency of access as well as the cost for each one to determine the right ordering.

We want to avoid any unnecessary refreshing from being scheduled, since it would waste CPU utilization without any benefit to the applications. One simple method is to have the master node send a special reply (e.g., $nochange$), when a slave node pulled the value of R for r_i, if the current value of R is the same as r_i. When a slave node receives $nochange$, it discards it so that the CPU utilization is not wasted. This idea can be extended further by introducing the notion of imprecise computation [9]. For each replicated data, we may specify the tolerance level of imprecision (R_p). If the change in the value of R is not more than R_p, a slave node will receive $nochange$. The proper value for R_p depends on the semantics of the data and applications that access it. We can customize the value of R_p, depending on how each copy is used by applications at each slave node. If r_i is used by an application whose R_p is different from that of applications that use r_j, the slave node that maintains r_i should specify its R_p when r_i is created.

Depending on the application semantics, real-time data services may need alert rules to be included for exception handling. The master node of a replicated data may have to push the data upon a certain condition, although the system is relying on the pull model. For example, consider the following situation. In a smart building, the temperature of a vacant room is pulled less frequently when the system has a transient overload. However, an alert can be triggered at the master node if the temperature exceeds a certain threshold (i.e., a possibility of fire). In that case, the master node should push its value to all the slave nodes with secondary copies.

6 Related Work

A novel on-line update scheduling policy has been proposed in the context of the web server [7]. The performance of a web server can be improved by caching dynamically generated data at the web server and the back-end database continuously updates them. Given the views to materialize, the proposed update scheduling policy can significantly improve the data freshness compared to FIFO scheduling. They discuss a complementary problem in [6], i.e., view selection problem to materialize. Trade-off issues between response time and data freshness are considered in their work. However, they provide neither miss ratio nor data freshness guarantees.

Stanford Real-Time Information Processor (STRIP) addressed the problem of balancing between the freshness and transaction timeliness [1]. In a real-time database, data should be maintained fresh to correctly reflect the status of the real-world environment, and transactions should be processed in a timely manner. To study the trade-off between freshness and timeliness, four scheduling algorithms were introduced to schedule updates and transactions, and the performance was compared. In their later work, a similar trade-off problem was studied for derived data [2].

An algorithm for update scheduling was proposed in a recent [7]. Their goal is to achieve the maximum overall QoD (quality of data), which is the weighted sum of the freshness probabilities of all views in the database. They have demonstrated the importance of the update scheduling problem using server log traces of trade and quote database from New York Stock Exchange. Unlike their approach, we provide a certain guarantee on deadline miss ratio. In this way, user service requests can be processed in a timely manner using fresh data.

Recently, feedback control has been widely applied to QoS management and real-time scheduling [12, 13, 18]. However, to our best knowledge none of them considered QoS management issues in real-time data services considering the timeliness and data freshness constraints.

7 Conclusions

The demand for real-time data services is increasing in web-based information systems and e-commerce applications. In these applications, it is desirable to process user service requests in time using fresh data. However, it is very challenging to satisfy this fundamental requirement due to possibly time-varying workloads and data access patterns in an open environment such as World Wide Web. In this paper, we presented a QoS management scheme to support real-time data services. In our approach, user requests can be processed in time using fresh data even in the presence of unpredictable workloads and data access patterns. A novel adaptive update policy is applied together with feedback and admission control schemes to manage potential overloads. We have also discussed related issues such as derived data management, differentiated services, and replica control in a distributed environment. The significance of our work will become more evident as the demand for (and importance of) real-time data services increases in web-based information systems and e-commerce applications.

References

[1] B. Adelberg, H. Garcia-Molina, and B. Kao. Applying Update Streams in a Soft Real-Time Database System. In *ACM SIGMOD*, 1995.

[2] B. Adelberg, B. Kao, and H. Garcia-Molina. Database Support for Efficiently Maintaining Derived Data. In *ETDB*, 1996.

[3] J. B. et al. DataBlitz Storage Manager: Main Memory Database Performance for Critical Applications . In *ACM SIGMOD - Industrial Session: Database Storage Management*, 2000.

[4] K. D. Kang, S. H. Son, and J. A. Stankovic. Service Differentiation in Real-Time Main Memory Databases. In *the 5th IEEE International Symposium on Object-oriented Real-time Distributed Computing*, April 2002.

[5] K. D. Kang, S. H. Son, J. A. Stankovic, and T. F. Abdelzaher. A QoS-Sensitive Approach for Timeliness and Freshness Guarantees in Real-Time Databases. In *the 14th Euromicro Conference on Real-Time Systems*, June 2002.

[6] A. Labrinidis and N. Roussopoulos. Adaptive WebView Materialization. In *the Fourth International Workshop on the Web and Databases, held in conjunction with ACM SIGMOD*, May 2001.

[7] A. Labrinidis and N. Roussopoulos. Update Propagation Strategies for Improving the Quality of Data on the Web. In *the 27th International Conference on Very Large Data Bases (VLDB'01)*, Rome, Italy, September 2001.

[8] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1979.

[9] J. W. Liu, K. Lin, W. Shih, A. Yu, J. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computation. *IEEE Computer*, 24(5), 1991.

[10] C. Lu, T. Abdelzaher, J. Stankovic, and S. H. Son. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, May 2001.

[11] C. Lu, J. Stankovic, T. Abdelzaher, G. Tao, S. H. Son, and M. Marley. Performance Specifications and Metrics for Adaptive Real-Time Systems. In *Real-Time Systems Symposium*, Orlando, Florida, November 2000.

[12] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback Control Real-Time Scheduling: Framework, Modeling and Algorithms. *Journal of Real-Time Systems, Special Issue on Control-Theoretical Approaches to Real-Time Computing*, 2002. To appear.

[13] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated Caching Services; A Control-Theoretical Approach. In *the 21st International Conference on Distributed Computing Systems*, Phoenix, Arizona, April 2001.

[14] E. Pacitti and E. Simon. Update Propagation Strategies to Improve Freshness in Lazy Master Replicated Databases. *VLDB Journal*, 8:305–318, 2000.

[15] C. L. Phillips and H. T. Nagle. *Digital Control System Analysis and Design (3rd edition).* Prentice Hall, 1995.

[16] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1(2), 1993.

[17] S. H. Son and K. D. Kang. QoS Management in Web-based Real-Time Data Services. Technical Report CS-2002-08, Computer Science Department at University of Virginia, 2002.

[18] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole. A Feedback-driven Proportion Allocator for Real-Rate Scheduling. In *the third Symposium on Operating Systems Design and Implementation*, 1999.

[19] TimesTen Performance Software. *TimesTen White Paper*. Available in the World Wide Web, http://www.timesten.com/library/index.html, 2001.