Generalized Performance Management of Multi-Class Real-Time Imprecise Data Services *

Mehdi Amirijoo, Nicolas Chaufette
Dept. of Computer and Information Science
Linköping University, Sweden
meham@ida.liu.se

Jörgen Hansson
Software Engineering Institute
Carnegie Mellon University, USA
hansson@sei.cmu.edu.

Sang H. Son
Dept. of Computer Science
University of Virginia, Charlottesville, USA
son@cs.virginia.edu

Svante Gunnarsson
Dept. of Electrical Engineering
Linköping University, Sweden
svante@isy.liu.se

Abstract

The intricacy of real-time data service management increases mainly due to the emergence of applications operating in open and unpredictable environments, increases in software complexity, and need for performance guarantees. In this paper we propose an approach for managing the quality of service of real-time databases that provide imprecise and differentiated services, and that operate in unpredictable environments. Transactions are classified into service classes according to their level of importance. Transactions within each service class are further classified into subclasses based on their quality of service requirements. This way transactions are explicitly differentiated according to their importance and quality of service requests. The performance evaluation shows that during overloads the most important transactions are guaranteed to meet their deadlines and that reliable quality of service is provided even in the face of varying load and execution time estimation errors.

1 Introduction

The demand for real-time data services increases due to the emergence of data intensive applications, e.g., engine control, web servers, and e-commerce. Further, these applications are becoming increasingly sophisticated in their real-time data needs, making ad hoc data management very difficult. Real-time databases (RTDBs) [22] have been introduced to address the problems arising in data management for real-time systems. In open systems, where exact execution time estimates, arrival rates, and data access patterns are not available, the workload of RTDBs cannot be precisely predicted and, hence, the databases can become overloaded. This results in uncontrolled deadline misses and freshness violations during transient overloads.

Several approaches, e.g., [15, 3] have been proposed for providing performance or quality of service (QoS) guarantees for real-time data services, despite the presence of inaccurate workload characteristics. QoS is defined in terms of deadline miss ratio, utilization, and the precision of data and transaction results. The approach called Robust Quality Management of Differentiated Imprecise Data Services (RDS) [3], was presented for managing the performance of differentiated and imprecise real-time data services. Data imprecision, i.e., allowing data objects in an RTDB to deviate from their corresponding real world values, and imprecise computation [18] have been used to trade off quality of transaction results versus the load applied on the system. In the approach taken by RDS, transactions are first classified according to their importance. The QoS requirement of each class is then chosen independently of the importance of the class. However, a serious restriction of RDS is that transactions with the same importance must use data of similar precision although the transactions may have different precision requirements on the data. Further, transactions in the same service class are required to deliver results of equal quality or precision. This is not general enough as applications submitting transactions may have the same level of importance, but different QoS requirements.

*This work was funded, in part by CUGS (the National Graduate School in Computer Science, Sweden), CENIIT (Center for Industrial Information Technology) under contract 01.07, NSF grants IIS-0208578 and CCR-0329609, and ISIS (Information Systems for Industrial Control and Supervision).

IEEE COMPUTER SOCIETY

Let us take the example of embedded RTDBs for engine control in automobiles [11].[1] Some of the control applications in automobiles are more important than others and, consequently, they must meet their deadlines. However, two equally important control applications may not require sensor data that are equally precise. In fact, one of the control applications may tolerate less accurate sensor data than the other one. As a consequence, the two applications submit transactions, which are equally important but have different QoS requirements. Previous approaches (e.g., [6, 15, 12]) and RDS do not allow equally important transactions to have different QoS requirements. Clearly, these transaction requirements imposed by applications must be met by RTDBs that provide real-time data services to the applications.

In this paper we present a generalized performance management scheme for multiclass real-time data services that captures the above mentioned requirements of applications, namely, equally important transactions may have different QoS requirements, which are expressed as data precision and transaction precision. The first contribution of this paper is a performance specification model that enables the applications to be classified according to their importance and QoS requirements. The specification model allows the transactions to be classified into service classes representing importance levels, and within each service class, transactions are divided into subclasses giving the QoS requirements of the transactions. The expressive power of the performance specification model allows a database operator[2] or database designer to specify the desired nominal system performance, and the worst-case system performance and system adaptability in the face of unexpected failures or load variation. The second contribution is an architecture and an algorithm, based on feedback control scheduling [21, 19, 2], for managing the QoS. Performance studies show that the suggested approach fulfills the performance requirements. Transaction and data precision are managed, even for transient overloads and with inaccurate execution time estimates of the transactions. We show that during overloads transactions are rejected in a strictly hierarchical fashion based on their service class, representing the importance of the transactions. Within a service class, the rejection rate is fairly distributed among the subclasses.

The rest of this paper is organized as follows. A problem formulation is given in section 2. In section 3, the assumed database model is given. In section 4, we present an approach for QoS management and in section 5, the results of performance evaluations are presented. In section 6, we give an overview on related work, followed by section 7, where conclusions and future work are discussed.

[1]The development of RTDBs for automobiles has been carried out together with our collaborators Fiat-GM.

[2]By a database operator we mean an agent, human or computer, that supervises and operates the database, including setting the QoS.

2 Problem Formulation

In our database model, data objects in an RTDB are updated by update transactions, e.g., sensor values, while user transactions represent user requests, e.g., complex read-write operations. We apply the notion of imprecision at user transaction level and data object level. We introduce the notion of transaction error (denoted te_i), inherited from the imprecise computation model [18], to measure the precision of the result of a user transaction. A transaction T_i returns more precise results, i.e., lower te_i, as it receives more processing time. We say that the quality of user transaction (QoT) increases as the transaction error of the user transactions decreases. Further, for a data object stored in the RTDB and representing a real-world variable, we can allow a certain degree of deviation compared to the real-world value. If such a deviation can be tolerated, arriving updates may be discarded during transient overloads, decreasing quality of data (QoD) as the imprecision of the data objects increases. To measure QoD we introduce the notion of data error (denoted de_i), which gives how much the value of a data object d_i stored in the RTDB deviates from the corresponding real-world value, which is given by the latest arrived transaction updating d_i. Note that the latest arrived transaction updating d_i may have been discarded and, hence, d_i may hold the value of an earlier update transaction. We define QoS in terms of QoT and QoD, i.e., the precision of the transaction results and the precision of the data stored in the RTDB.

Assume that there are V service classes, where $SVC = \{svc^1, \ldots, svc^v, \ldots, svc^V\}$ $(1 \leq v \leq V)$ denotes the set of service classes. Each service class svc^v holds B_v subclasses, namely $svc^v = \{sbc^{v,1}, sbc^{v,2}, \ldots, sbc^{v,b_v}, \ldots, sbc^{v,B_v}\}$, where $1 \leq b_v \leq B_v$. Let B denote the total number of subclasses, i.e., $B = \sum_{v=1}^{V} B_v$. User transactions are classified into service classes based on their importance. In a service class where transactions are equally important, transactions are further divided into subclasses. Each subclass represents the unique QoS requirement of the transactions in that subclass. Transactions in $sbc^{1,1}, \ldots, sbc^{1,B_1}$ are the most important transactions, and transactions in $sbc^{2,1}, \ldots, sbc^{2,B_2}$ are less important and so on. In general transactions in sbc^{v,b_v} are more important than transactions in $sbc^{v',b_{v'}}$ if and only if $v < v'$, for any b_v and $b_{v'}$. Transactions in sbc^{v,b_v} and $sbc^{v',b_{v'}}$ are equally important if and only if $v = v'$. For a given service class svc^v, transactions are divided into the subclasses $sbc^{v,1}, \ldots, sbc^{v,b_v}, \ldots, sbc^{v,B_v}$ according to their QoS requirements. For a subclass sbc^{v,b_v}, the desired QoS is expressed in terms of te_i that the transactions in sbc^{v,b_v} produce and the data error of the data objects that transactions in sbc^{v,b_v} read. Subclasses of a service class hold transactions that are equally important but

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

that have different QoS requirements. In particular, for any b'_v where $1 \leq b'_v \leq B_v$ and $b'_v \neq b_v$, transactions in sbc^{v,b'_v} and sbc^{v,b_v} are equally important, however, they have different QoS requirements.

The problem that we pose in this work is to find a performance specification model for expressing the importance and the QoS requirements, in terms of data error and transaction error, of the subclasses. The performance specification model must capture the independence between importance and QoS requirements, and allow several QoS requirements to be specified for a set of equally important transactions. Further, an architecture and a set of algorithms that respect the importance of the transactions and that manage data error and transaction error such that given QoS specifications for each subclass are satisfied, must be developed.

3 Data and Transaction Model

We consider a main memory database model, where there is one CPU as the main processing element. We consider the following data and transaction models. In our data model, data objects can be classified into two classes, temporal and non-temporal [22]. For temporal data we only consider base data, i.e., data objects that hold the view of the real-world and are updated by sensors. A base data object d_i is considered temporally inconsistent or stale if the current time is later than the timestamp of d_i followed by the absolute validity interval avi_i of d_i, i.e., $currenttime > timestamp_i + avi_i$. For a data object d_i, let data error $de_i = \Phi(cv_i, v_j)$ be a non-negative function of the current value cv_i of d_i and the value v_j of the latest arrived transaction that updated d_i or that was to update d_i but was discarded. The function Φ may for example be defined as the absolute deviation between cv_i and v_j, i.e., $de_i = |cv_i - v_j|$, or the relative deviation as given by $de_i = \frac{|cv_i - v_j|}{|cv_i|}$. To capture the QoD demands of the different service classes we model the data error as perceived by the transactions in sbc^{v,b_v} with $de_i \times def^{v,b_v}$ where def^{v,b_v} denotes the data error factor of the transactions in sbc^{v,b_v}. The greater def^{v,b_v} is, the greater does the transactions in sbc^{v,b_v} perceive the data error. We define the weighted data error as $wde_i = de_i \times def_{d,i}$, where $def_{d,i}$ is the maximum data error factor of the transactions accessing d_i.

Update transactions arrive periodically and may only write to base data objects. User transactions arrive aperiodically and may read temporal and read/write non-temporal data. User and update transactions (T_i) are composed of one mandatory subtransaction m_i and $|O_i| \geq 0$ optional subtransactions $o_{i,j}$, where $o_{i,j}$ is the j^{th} optional subtransaction of T_i. For the remainder of the paper, we let $t_{i,j}$ denote the j^{th} subtransaction of T_i. Since updates do not use complex logical or numerical operations, we assume that each update transaction consists only of a single mandatory subtransaction, i.e., $|O_i| = 0$. We use the milestone approach [18] to transaction imprecision. Thus, we divide transactions into subtransactions according to milestones. A mandatory subtransaction is completed when it is completed in a traditional sense. The mandatory subtransaction gives an acceptable result and must be computed to completion before the transaction deadline. The optional subtransactions may be processed if there is enough time or resources available. While it is assumed that all subtransactions of a transaction T_i arrive at the same time, the first optional subtransaction (if any) $o_{i,1}$ becomes ready for execution when the mandatory subtransaction, m_i, is completed. In general, an optional subtransaction, $o_{i,j}$, becomes ready for execution when $o_{i,j-1}$ (where $2 \leq j \leq |O_i|$) completes. We set the deadline of every subtransaction $t_{i,j}$ to the deadline of the transaction T_i. A subtransaction is terminated if it has completed or has missed its deadline. A transaction T_i is terminated when $o_{i,|O_i|}$ completes or one of its subtransactions misses its deadline. In the latter case, all subtransactions that are not completed are terminated as well.

For a user transaction T_i, we use an error function [7] to approximate its transaction error given by,

$$te_i(|COS_i|) = \left(1 - \frac{|COS_i|}{|O_i|}\right)^{n_i} \quad (1)$$

where n_i is the order of the error function and $|COS_i|$ denotes the number of completed optional subtransactions. By choosing n_i we can model and support multiple types of transactions showing different error characteristics.

4 Approach

Below we describe an approach for managing the performance of an RTDB in terms of transaction and data quality. First, we start by defining QoS and how it can be specified. An overview of the feedback control scheduling architecture is given, followed by issues related to modeling of the architecture and design of controllers. We refer to the presented approach as Management of Multi Class Real-Time Imprecise Data Services (MCIDS).

4.1 Performance Metrics and QoS specification

We apply the following steady state and transient state performance metrics [19] to each service class. The set $Terminated^{v,b_v}(k)$ denotes the transactions in subclass sbc^{v,b_v} that are terminated during the interval $[(k-1)T, kT]$, where T is the sampling period. For the rest of this paper, we sometimes drop k where the notion of time is not important.

- The average transaction error of admitted user transac-

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

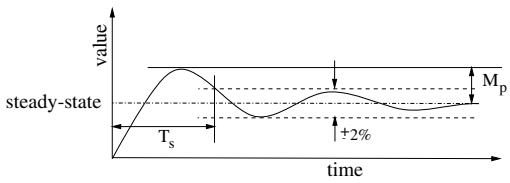


Figure 1. Definition of settling time (T_s) and overshoot (M_p)

tions,

$$ate^{v,b_v}(k) = 100 \times \frac{\sum_{i \in Terminated^{v,b_v}(k)} te_i}{|Terminated^{v,b_v}(k)|}(\%)$$

gives the precision of the results produced by user transactions.

- The data precision requirement of user transactions is given using def^{v,b_v}.
- Data precision is manipulated by managing the data error of the data objects, which is done by considering an upper bound for the weighted data error given by the maximum weighted data error $mwde$. An update transaction T_j is discarded if the weighted data error of the data object d_i to be updated by T_j is less or equal to $mwde$ (i.e., $wde_i \leq mwde$). If $mwde$ increases, more update transactions are discarded, degrading the quality of data. Setting $mwde$ to zero results in the highest data precision, while setting $mwde$ to one results in lowest data precision allowed.
- Overshoot M_p^{v,b_v} is the worst-case system performance in the transient system state (see Figure 1) and it is given in percentage. Overshoot is specified for ate^{v,b_v}.
- Settling time T_s^{v,b_v} is the time for the transient overshoot to decay and reach the steady state performance (see Figure 1), hence, it is a measure of system adaptability, i.e., how fast the system converges toward the desired performance. Settling time is specified for ate^{v,b_v}.
- Admission Percentage, $ap^{v,b_v}(k) = 100 \times \frac{|Admitted^{v,b_v}(k)|}{|Submitted^{v,b_v}(k)|}(\%)$, where $|Admitted^{v,b_v}(k)|$ is the number of admitted transactions and $|Submitted^{v,b_v}(k)|$ is the number of submitted transactions in sbc^{v,b_v}.

We define QoD in terms of $mwde$ and an increase in QoD refers to a decrease in $mwde$, while a decrease in QoD refers to an increase in $mwde$. Similarly, we define QoT for a subclass sbc^{v,b_v} in terms of ate^{v,b_v}. QoT for a subclass sbc^{v,b_v} increases as ate^{v,b_v} decreases, while QoT decreases as ate^{v,b_v} increases. The QoS specification is given in terms of def^{v,b_v} and a set of target levels in the steady state or references ate_r^{v,b_v} for ate^{v,b_v}.

Turning to the QoD requirement specification, assume we want that $de_i \leq \xi^{v,b_v}$ for a subclass sbc^{v,b_v}, where ξ^{v,b_v} is an arbitrary data error. We observe that a data object may be accessed by several transactions in different subclasses and, hence, there may be different precision requirements put upon the data item. It is clear that we need to ensure that the data error of the data object complies with the needs of all transactions and, consequently, any data error conflicts must be resolved by satisfying the needs of the transaction with the stronger requirement. Assume that several transactions with different precision requirements access d_i. Then for all v and b_v it must hold that $def^{v,b_v} \leq def_{d,i}$, since $def_{d,i}$ is the maximum data error factor of the transactions accessing d_i. The data object d_i is least precise when $mwde$ is equal to one and, hence, $de_i \times def^{v,b_v} \leq de_i \times def_{d,i} = wde_i \leq 1$. From this we conclude that $de_i \leq \frac{1}{def^{v,b_v}}$. So by setting def^{v,b_v} to $\frac{1}{\xi^{v,b_v}}$ we satisfy the QoD requirement of the transactions in sbc^{v,b_v}.

The following example shows a specification of QoS requirements: $\{ate_r^{1,1} = 30\%, def^{1,1} = 1, T_s^{1,1} \leq 60s, M_p^{1,1} \leq 35\%\}$, $\{ate_r^{1,2} = 40\%, def^{1,2} = 0.3, T_s^{1,2} \leq 60s, M_p^{1,2} \leq 35\%\}$, $\{ate_r^{2,1} = 20\%, def^{2,1} = 3, T_s^{2,1} \leq 60s, M_p^{2,1} \leq 35\%\}$, $\{ate_r^{2,2} = 50\%, def^{2,2} = 0.1, T_s^{2,2} \leq 60s, M_p^{2,2} \leq 35\%\}$, $\{ate_r^{3,1} = 10\%, def^{3,1} = 1, T_s^{3,1} \leq 60s, M_p^{3,1} \leq 35\%\}$. Note that transactions in $sbc^{1,1}$ and $sbc^{1,2}$ are equally important, but they have different QoS requirements. Also, transactions in $sbc^{1,1}$ are more important than transactions in $sbc^{2,1}$, however, the QoS requirement for $sbc^{1,1}$ is weaker than the QoS requirement for $sbc^{2,1}$, i.e., $ate_r^{1,1} > ate_r^{2,1}$ and $def^{1,1} < def^{2,1}$. This shows that several QoS levels may be associated with a certain importance level and that the specification of importance and QoS requirements are orthogonal.

4.2 Data Precision Classes

We need to make sure to meet the data precision requirement of the transactions in all subclasses. We take the approach presented in [3], i.e., data objects are classified according to the precision requirement of the transactions accessing them, where each class of data objects represents a data precision requirement. When a user transaction with a very high data precision requirement accesses a data object, we classify the data object to a higher data precision class representing greater precision. In particular, once a transaction T_i accesses a data object d_i where the data error factor def^{v,b_v} of T_i is greater than the current data error factor $def_{d,i}$ of d_i, then we set $def_{d,i}$ to def^{v,b_v}. If after

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

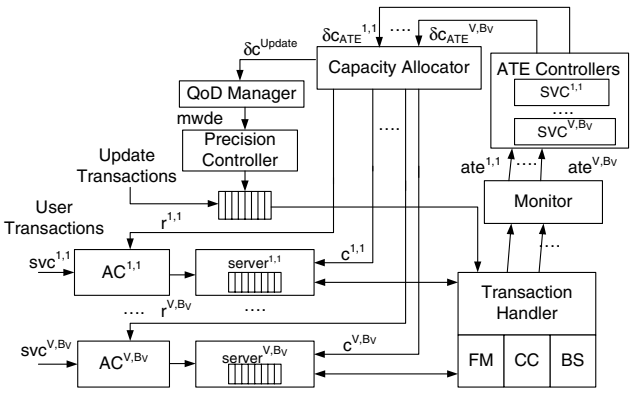


Figure 2. QoS management architecture using feedback control

a while, no transaction with equal or greater precision requirement accesses that data object, then we move the data object to a lower data precision class, representing a lower precision requirement. This corresponds to the data error factor $def_{d,i}$ decreasing linearly over time, moving to lower data precision classes until a transaction with a higher data error factor accesses d_i. This way the system is adaptive to changes in access patterns.

4.3 Feedback Control Scheduling Architecture

The architecture of our QoS management scheme is given in Figure 2. Update transactions have higher priority than user transactions and are upon arrival ordered in an update ready queue according to the earliest deadline first (EDF) scheduling policy (for an elaborate discussion on EDF see e.g., [6]). To provide individual QoS guarantees for each user transaction subclass we have to enforce isolation among the subclasses by bounding the total execution time of the transactions in each subclass. This is achieved by using servers [6], which enable us to limit the resource consumption of transactions, while lower average response time is enforced for the transactions in higher service classes. Let $server^{v,b_v}$ denote the server for sbc^{v,b_v} and c^{v,b_v} denote the capacity (in terms of transaction execution time) of $server^{v,b_v}$. We assign priorities to the servers according to their importance, i.e., $server^{v,b_v}$ has higher priority than $server^{v+1,b_v}$. Servers within a service class have the same priority, i.e., $server^{v,1}, \ldots, server^{v,B_v}$ have the same priority.

At the beginning of the sampling interval, $server^{1,1}$ serves any pending user transactions in its ready queue within the limit of $c^{1,1}$ or until no more user transactions are waiting, at which point $server^{1,1}$ becomes suspended and the following server in the same service class, i.e., $server^{1,2}$ becomes active (if it exists). This procedure is continued until $server^{1,B_1}$ becomes suspended, at which point the server of $sbc^{2,1}$, i.e., $server^{2,1}$ becomes active. At any point in time $server^{2,b_2}$ may be interrupted and suspended, and $server^{1,b_1}$ is then reactivated if new user transactions in sbc^{1,b_1} arrive and $server^{1,b_1}$ has any capacity left. In general, $server^{v,b_v}$ serves any pending user transactions in its ready queue within the limit of c^{v,b_v} or until no more user transactions are waiting, at which point $server^{v,b_v}$ becomes suspended and the next server in svc^v, i.e., $server^{v,b_v+1}$ becomes active. If $server^{v,b_v}$ is the last server of svc^v (i.e., $b_v = B_v$), then $server^{v+1,1}$ becomes active. The server $server^{v',b_{v'}}$ is suspended and $server^{v,b_v}$ is reactivated if and only if $v < v'$, new user transactions in sbc^{v,b_v} arrive, and c^{v,b_v} is greater than zero. The capacity is replenished periodically with the sampling period T.

The transaction handler manages the execution of the transactions. It consists of a freshness manager (FM), a unit managing the concurrency control (CC), and a basic scheduler (BS). The FM checks the freshness before accessing a data object, using the timestamp and the absolute validity interval of the data. We employ two-phase locking with highest priority (2PL-HP) [1] for concurrency control. 2PL-HP is chosen since it is free from priority inversion and has well-known behavior. EDF, where transactions are processed in the order determined by their absolute deadlines, is used as a basic scheduler. To ensure that the mandatory subtransactions meet their deadlines, the mandatory subtransactions have higher priority than the optional subtransactions.

At each sampling instant k, the controlled variables ate^{v,b_v} are monitored and fed into the ATE controllers, which compare the performance references ate_r^{v,b_v} with ate^{v,b_v} to get the current performance errors. Based on this each ATE controller computes a requested change $\delta c_{ATE}^{v,b_v}$ to c^{v,b_v}. If ate^{v,b_v} is higher than ate_r^{v,b_v}, then a positive $\delta c_{ATE}^{v,b_v}$ is returned, requesting an increase in the capacity so that ate^{v,b_v} is lowered to its reference. The requested changes in capacities are given to the capacity allocator, which distributes the capacities according to the service class level. During overloads it may not be possible to accommodate all requested capacities. Instead, the QoD is lowered, resulting in more discarded update transactions, hence, more resources can be allocated for user transactions. If the lowest data quality is reached and no more update transactions can be discarded, the amount of capacity r^{v,b_v} that is not accommodated is returned to the admission controller, which rejects transactions with a total execution time of r^{v,b_v}.

For the purpose of the controller design we have modeled the controlled system using $\mathcal{Z}$-transform theory [8]. Starting with the manipulated variable, the capacity in the next period is,

$$c^{v,b_v}(k+1) = c^{v,b_v}(k) + \delta c_{ATE}^{v,b_v}(k), \qquad (2)$$

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

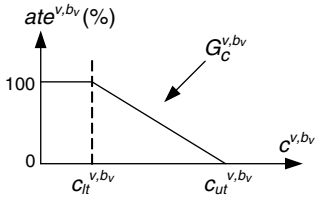


Figure 3. The relation between capacity and average transaction error

i.e., the capacity is the integration over the manipulated variable. Now, there exists a nonlinear relation between c^{v,b_v} and ate^{v,b_v}, as shown in Figure 3. For capacities less than the lower threshold c_{lt}^{v,b_v} none of the optional subtransactions are completed and according to (1) the transaction error of the user transactions is one, hence, $ate^{v,b_v} = 100\%$. The average transaction error ate^{v,b_v} decreases as c^{v,b_v} increases, since more CPU time is allocated to user transactions in sbc^{v,b_v}. There exists an upper threshold c_{ut}^{v,b_v} at which ate^{v,b_v} becomes zero as all optional subtransactions are completed. We linearize the relationship between ate^{v,b_v} and c^{v,b_v} at the vicinity of ate_r^{v,b_v}, i.e.,

$$ate^{v,b_v}(k+1) = ate^{v,b_v}(k) + G_c^{v,b_v}(c^{v,b_v}(k+1) - c^{v,b_v}(k)) \quad (3)$$

where G_c^{v,b_v} is the ate gain, giving the derivative of the function relating ate^{v,b_v} and c^{v,b_v} (see Figure 3). Inserting (2) into (3) gives,

$$ate^{v,b_v}(k+1) = ate^{v,b_v}(k) + G_c^{v,b_v} \times \delta c_{ATE}^{v,b_v}(k). \quad (4)$$

For simplicity we use the same ate gain for all subclasses, hence, we denote the ate gain of the subclasses with G_c. By taking the $\mathcal{Z}$-transform of (4), we obtain the transfer function,

$$P^{v,b_v}(z) = \frac{ate^{v,b_v}(z)}{\delta c_{ATE}^{v,b_v}(z)} = \frac{G_c}{z-1},$$

describing ate^{v,b_v} in terms of $\delta c_{ATE}^{v,b_v}$. We have tuned G_c by measuring ate for different capacities and taking the slope at ate_r. The ATE controller is implemented using a P-controller, i.e., $\delta c_{ATE}^{v,b_v}(k) = K_P(ate_r^{v,b_v}(k) - ate^{v,b_v}(k))$, where the controller parameter K_P is tuned with root locus [8].

4.4 Data and Transaction Error Management

In the following sections we present an algorithm used for computing the capacity of the servers and a method for controlling the precision of the data.

4.4.1 Capacity Allocation

Figure 4 shows how c^{v,b_v}, r^{v,b_v}, and $mwde$ are computed. We start by describing **ComputeCapacity**, which implements the capacity allocator in Figure 2. Although there is no server for update transactions, we simplify the presentation of the algorithm by letting the capacity of the update transactions denote the execution time of update transactions. Let $c^{Update}(k)$ denote the measured capacity, or equivalently the execution time, of the update transactions. As the arrival patterns of update transactions are varying, we take the moving average $c_{MA}^{Update}(k)$ of the update transaction capacity to smoothen out great variations (line 1).

We start allocating capacities with respect to the service classes, starting with subclasses in svc^1. The requested capacity $c_{req}^{v,b_v}(k+1)$ of subclass sbc^{v,b_v} is the sum of the previously assigned capacity $c^{v,b_v}(k)$ and the requested change in capacity $\delta c_{ATE}^{v,b_v}(k+1)$ that is computed by the ATE controller (lines 5-7). Then we compute the sum $c_{req}^{v}(k+1)$ of the requested capacities of all subclasses of service class svc^v (line 8). If the total available capacity, given by $T - c(k+1)$, is less than the requested capacity $c_{req}^{v}(k+1)$ then we degrade QoD by calling **ChangeUpdateC** along with how much update capacity to free (lines 9-11). See Section 4.4.2 for an elaborate description on **ChangeUpdateC**. Then we compute a ratio $ratio^v$ giving how much of the requested capacity can be allocated (lines 12-16). The assigned capacities are computed by taking the product of the ratio and the requested capacities (lines 17-20).

If the entire requested capacity cannot be accommodated, i.e., $ratio^v < 1$, we enforce the capacity adjustment by rejecting more transactions (lines 21-24). One key concept in the capacity allocation algorithm is that we employ an optimistic admission policy. We start by admitting all transactions and in the face of an overload we reject transactions until the overload is resolved. This contrasts against pessimistic admission policy where the admission percentage is increased as long as the system is not overloaded. We believe that the pessimistic admission policy is not suitable as some critical transactions, i.e., the transactions in higher service classes, are initially rejected. Continuing with the algorithm description, if the requested capacity is accommodated then we try to reduce the number of rejected transactions (lines 25-33). This is done by first trying to degrade QoD as much as possible in order to accommodate as many user transactions as possible (lines 26-28). Then the number of rejected transactions is lowered and additional capacity is allocated to compensate for the decrease in number of rejected transactions (lines 29-33). If the requested capacity is accommodated and no transactions were rejected during the previous sampling interval then we do not reject any transactions during the next sampling interval (lines 35-37). Finally, after the capacity allocation we check to see whether

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

there is any spare capacity $c_s(k+1)$, and if so we upgrade QoD within the limits of $c_s(k+1)$ (line 42). If QoD cannot be further upgraded, i.e., $mwde = 0$, then we distribute $c_s(k+1)$ among the servers (lines 43-47).

The **ComputeCapacity** algorithm in Figure 4 runs in the worst case in $\mathcal{O}(VB_{max})$, where B_{max} is the maximum number of subclasses in a service class, i.e., $B_{max} = \max_{1 \le v \le V} B_v$. As is noted, the time complexity $\mathcal{O}(VB_{max})$ is pseudo-polynomial [9]. Since the maximum number of subclasses B_{max} is bounded, the capacity allocation algorithm is linear in V. Similarly, as the number of service classes V is bounded, the algorithm is linear in B_{max}. This shows that the algorithm scales well with the number of service classes and number of subclasses.

4.4.2 QoD Management

We now study the algorithm **ChangeUpdateC**. The precision of the data is controlled by the QoD Manager by setting $mwde(k)$ depending on the requested change in update capacity $\delta c^{Update}(k)$, see Figure 2. Rejecting an update results in a decrease in update capacity. Let $Discarded(k)$ be the set of discarded update transactions during interval $[(k-1)T, kT]$ and eet_i be the estimated execution time of update transaction T_i. We define the gained capacity, $gc(k) = \sum_{T_i \in Discarded(k)} eet_i$, as the capacity gained due to the result of rejecting one or more updates during interval $[(k-1)T, kT]$. In our approach, we profile the system and measure gc for different $mwde$s and linearize the relationship between these two, i.e., $mwde(k) = \mu(k) \times gc(k)$. Further, since RTDBs are dynamic systems such that the behavior of the system and environment is changing, the relation between $gc(k)$ and $mwde(k)$ is adjusted on-line. This is done by measuring $gc(k)$ for a given $mwde(k)$ during each sampling period and updating $\mu(k)$. Having the relationship between $gc(k)$ and $mwde(k)$, we introduce the function $h(\delta c^{Update}(k)) = \mu(k) \times (gc(k) - \delta c^{Update}(k))$, which returns an $mwde$ given $\delta c^{Update}(k)$. Since $mwde(k)$ cannot be greater than one and less than zero we use the function,

$$f(\delta c^{Update}(k)) = \begin{cases} 1, & h(\delta c^{Update}(k)) > 1 \\ h(\delta c^{Update}(k)), & 0 \le h(\delta c^{Update}(k)) \le 1 \\ 0, & h(\delta c^{Update}(k)) < 0 \end{cases} \quad (5)$$

to enforce this requirement. Now that we have arrived at f it is straightforward to compute $mwde$, as shown in Figure 4. The function **ChangeUpdateC** returns the estimated change in update capacity given the requested change $\delta c^{Update}(k)$.

ComputeCapacity($\delta c^{1,1}_{ATE}(k+1), \ldots, \delta c^{V,B_V}_{ATE}(k+1)$)

1: $c^{Update}_{MA}(k) \leftarrow \alpha \times c^{Update}(k) + (1-\alpha) \times c^{Update}_{MA}(k-1)$
2: $c(k+1) \leftarrow c^{Update}_{MA}(k)$
3: **for** $v = 1$ to V **do**
4: $r^v(k) \leftarrow \sum_{b_v=1}^{B_v} r^{v,b_v}(k)$
5: **for** $b_v = 1$ to B_v **do**
6: $c^{v,b_v}_{req}(k+1) \leftarrow max(0, c^{v,b_v}(k) + \delta c^{v,b_v}_{ATE}(k+1))$
7: **end for**
8: $c^v_{req}(k+1) \leftarrow \sum_{b_v=1}^{B_v} c^{v,b_v}_{req}(k+1)$
9: **if** $T - c(k+1) < c^v_{req}(k+1)$ **then** {if the total available capacity if less than the requested capacity}
10: $c(k+1) \leftarrow c(k+1)$ + **ChangeUpdateC**$(T - c(k+1) - c^v_{req}(k+1))$
11: **end if**
12: **if** $c^v_{req}(k+1) > 0$ **then**
13: $ratio^v = \frac{min(T-c(k+1), c^v_{req}(k+1))}{c^v_{req}(k+1)}$
14: **else**
15: $ratio^v = 0$
16: **end if**
17: **for** $b_v = 1$ to B_v **do**
18: $c^{v,b_v}(k+1) \leftarrow c^{v,b_v}_{req}(k+1) \times ratio^v$
19: $c(k+1) \leftarrow c(k+1) + c^{v,b_v}(k+1)$
20: **end for**
21: **if** $ratio^v < 1$ and $c^v_{req}(k+1) > 0$ **then** {if the assigned capacity is less than the requested capacity}
22: **for** $b_v = 1$ to B_v **do**
23: $r^{v,b_v}(k+1) \leftarrow r^{v,b_v}(k) + c^{v,b_v}_{req}(k+1) - c^{v,b_v}(k+1)$
24: **end for**
25: **else if** $r^v(k) > 0$ **then** {if the requested capacity was accommodated and we rejected transactions during the previous period}
26: **if** $T - c(k+1) < r^v(k)$ **then** {if the available capacity is less than the rejected capacity in the previous period}
27: $c(k+1) \leftarrow c(k+1)$ + **ChangeUpdateC**$(T - c(k+1) - r^v(k))$
28: **end if**
29: **for** $b_v = 1$ to B_v **do**
30: $r^{v,b_v}(k+1) \leftarrow \frac{max(0, r^v(k) - T + c(k+1)) \times r^{v,b_v}(k)}{r^v(k)}$
31: $c^{v,b_v}(k+1) \leftarrow c^{v,b_v}(k+1) + r^{v,b_v}(k) - r^{v,b_v}(k+1)$
32: $c(k+1) \leftarrow c(k+1) + r^{v,b_v}(k) - r^{v,b_v}(k+1)$
33: **end for**
34: **else**
35: **for** $b_v = 1$ to B_v **do**
36: $r^{v,b_v}(k+1) \leftarrow 0$
37: **end for**
38: **end if**
39: **end for**
40: $c_s(k+1) \leftarrow T - c(k+1)$
41: **if** $c_s(k+1) > 0$ **then** {if there is spare capacity}
42: $c_s(k+1) \leftarrow max(0, c_s(k+1) -$ **ChangeUpdateC**$(c_s(k+1)))$
43: **for** $v = 1$ to V **do**
44: **for** $b_v = 1$ to B_v **do**
45: $c^{v,b_v}(k+1) \leftarrow c^{v,b_v}(k+1) + \frac{c_s(k+1)}{B}$
46: **end for**
47: **end for**
48: **end if**

ChangeUpdateC($\delta c^{Update}(k)$)

1: $mwde(k+1) \leftarrow f(\delta c^{Update}(k))$
2: Return $\frac{1}{\mu}(mwde(k) - mwde(k+1))$

Figure 4. Data and Transaction Error Management Algorithms.

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

5 Performance Evaluation

5.1 Experimental Goals

The performance evaluation is undertaken by a set of simulation experiments, where a set of parameters have been varied. These are: (i) load ($load$), as computational systems may show different behavior for different loads, especially when the system is overloaded, and (ii) execution time estimation error ($esterr$), since often exact execution time estimates of transactions are not known. Execution time estimation errors induce additional unpredictability in the controlled variable, i.e., ate, as the transaction scheduling is based on inaccurate information.

5.2 Simulation Setup

The simulated workload consists of update and user transactions, which access data and perform virtual arithmetic/logical operations on the data. One simulation run lasts for 10 minutes of simulated time. For all the performance data, we have taken the average of 10 simulation runs and derived 95% confidence intervals. The workload models of the update and user transactions are described as follows. We use the following notation where the attribute x_i refers to the transaction T_i, and $x_i[t_{i,j}]$ is associated with the subtransaction $t_{i,j}$ of T_i.

Data and Update Transactions. The database holds 1000 temporal data objects (d_i) where each data object is updated by a stream ($stream_i$, $1 \leq i \leq 1000$). The period (p_i) is uniformly distributed in the range (100ms,50s), i.e., $U : (100ms, 50s)$, and the estimated execution time (eet_i) is given by $U : (1ms, 5ms)$. The average update value (av_i) of each $stream_i$ is given by $U : (0, 100)$. Upon a periodic generation of an update, $stream_i$ gives the update an actual execution time given by the normal distribution $N : (eet_i, \sqrt{eet_i})$ and a value (v_i) according to $N : (av_i, \sqrt{av_i \times varfactor})$, where $varfactor$ is uniformly distributed in $(0, 0.5)$. The deadline is set to $arrivaltime_i + p_i$. We define data error as the relative deviation between cv_i and v_j, i.e., $de_i = 100 \times \frac{|cv_i - v_j|}{|cv_i|}(\%)$.

User Transactions. Each $source_i$ generates a transaction T_i, consisting of one mandatory subtransaction and $|O_i|$, uniformly distributed between 1 and 10, optional subtransaction(s). The estimated (average) execution time of the mandatory and the optional ($eet_i[t_{i,j}]$) subtransactions is given by $U : (1ms, 4ms)$. The estimation error $esterr$ is used to introduce execution time estimation error in the average execution time given by $aet_i[t_{i,j}] = (1+esterr) \times eet_i[t_{i,j}]$. Further, upon generation of a transaction, $source_i$ associates an actual execution time to each subtransaction $t_{i,j}$, given by $N : (aet_i[t_{i,j}], \sqrt{aet_i[t_{i,j}]})$. The deadline is set to $arrivaltime_i + eet_i \times slackfactor$. The slack factor is uniformly distributed according to $U : (10, 20)$.

In the experiment presented here, we consider five subclasses $sbc^{1,1}$, $sbc^{1,2}$, $sbc^{2,1}$, $sbc^{2,2}$, and $sbc^{3,1}$. The QoS specification given in Section 4.1 is used. The workload submitted to the RTDB is distributed among the subclasses according to 25%, 25%, 10%, 25%, and 15%. The workload distribution captures the cases when the workload is equally divided among the subclasses in a service class (subclasses in svc^1) and the case when the workload is not equally divided among the subclasses in a service class (subclasses in svc^2). No workload is submitted to the RTDB before time $0s$, hence, the critical instant occurs at $0s$ which produces the worst case workload change. This puts the performance of the system in a transient state at $0s$, followed by a steady state once the workload has settled.

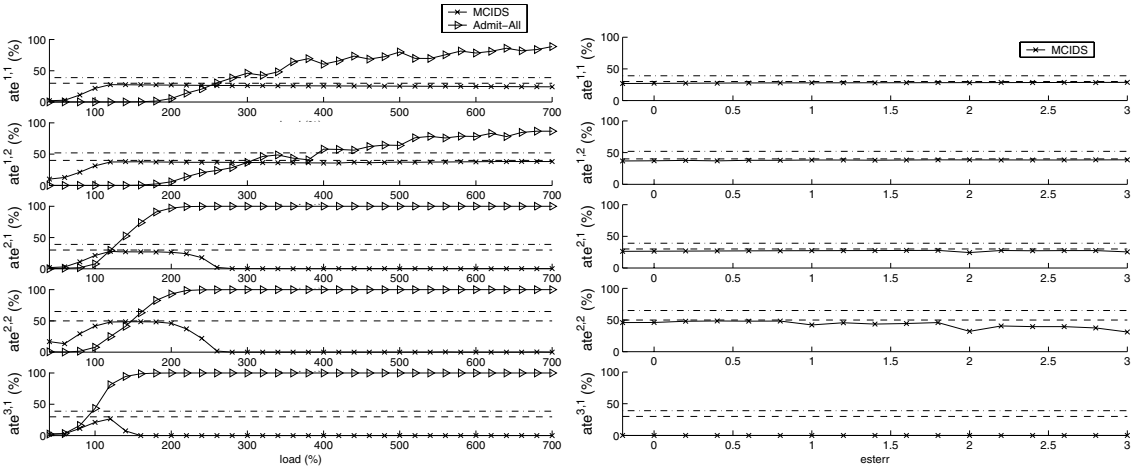
5.3 Baseline

To the best of our knowledge, the only work on techniques for managing data imprecision and transaction imprecision satisfying QoS or QoD requirements for differentiated services was presented by Amirijoo et al. [3]. However, that approach does not support multiple QoS requirements within a service class, and as such that approach and MCIDS are not comparable. For this reason we compare MCIDS with a baseline, called Admit-All, where all transactions are admitted, and no QoD management is performed. Transactions with equal importance are inserted in the same ready queue and scheduled using EDF. The ready queues are processed in the order of importance. This way we can study the impact of the workload on the system, e.g., how ate is affected by increasing workload.

5.4 Experiment 1: Results of Varying Load

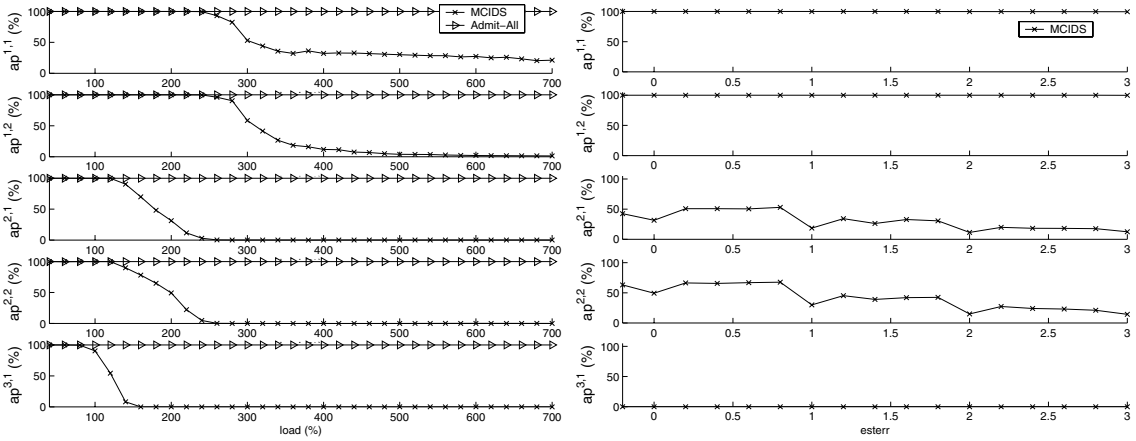
The goal of this experiment is to see how MCIDS reacts to increasing submitted load. We show that MCIDS satisfies a given QoS specification and that the importance requirements of the transactions are met. We measure ate, ap, and $mwde$ and apply loads from 40% to 700%. The execution time estimation error is set to zero (i.e. $esterr = 0$). We measure the system performance during steady state when the load is not changing considerably (the transient state performance is examined in detail in Section 5.6, where it is shown that MCIDS also manages significant variations in load). To measure the performance of the system during steady state, we start measuring ate, ap, and $mwde$ after $200s$ to remove the effects of the transients in the beginning of the simulation. Figures 5(a), 6(a), and 7(a) show ate, ap, and $mwde$ when $load$ is varied. The dashed lines indicate references, while the dashed-dotted lines give the specified overshoot (according to the QoS specification). The

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY



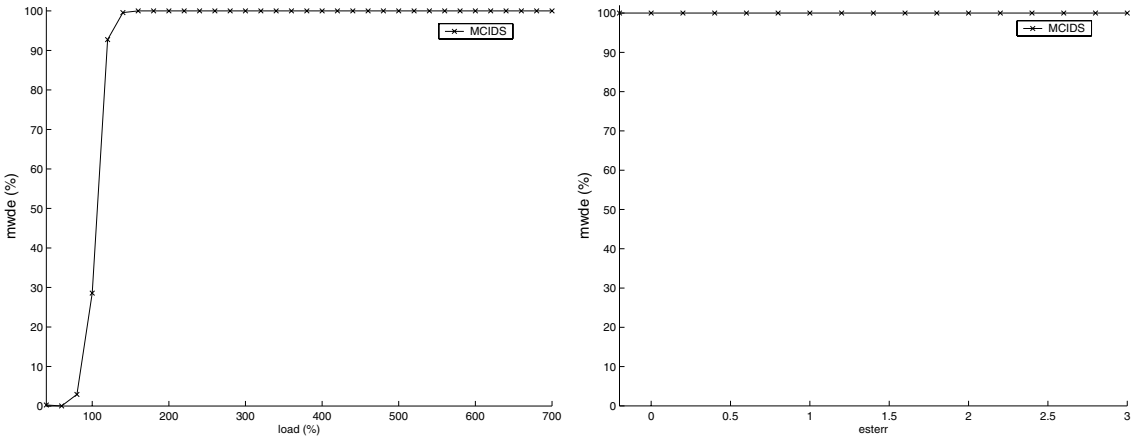
(a) Varying $load$ (b) Varying $esterr$

Figure 5. Average Transaction Error (ate)



(a) Varying $load$ (b) Varying $esterr$

Figure 6. Admission Percentage (ap)



(a) Varying $load$ (b) Varying $esterr$

Figure 7. Maximum Weighted Data Error ($mwde$)

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY

confidence intervals for ate, ap, and $mwde$ are less than $[ate - 6.81\%, ate + 6.81\%]$, $[ap - 13.69\%, ap + 13.69\%]$, and $[mwde - 17.02\%, mwde + 17.02\%]$, respectively.

Starting with the admission percentage given in Figure 6(a), we note that as the load increases the admission percentage $ap^{3,1}$ of subclass $sbc^{3,1}$, representing the least important transactions, decreases and more transactions are rejected. Transactions in $sbc^{2,1}$ and $sbc^{2,2}$ are the second least important transactions and, therefore, the admission percentage of $sbc^{2,1}$ and $sbc^{2,2}$, i.e., $ap^{2,1}$ and $ap^{2,2}$, starts decreasing when most of the transactions in $sbc^{3,1}$ are rejected. Similarly, the most important transactions, i.e., transactions in $sbc^{1,1}$ and $sbc^{1,2}$, are rejected when the less important transactions (transactions in $sbc^{2,1}$, $sbc^{2,2}$, and $sbc^{3,1}$) are rejected. Hence, the strict hierarchic admission policy, where the least important transactions are rejected in favor of the most important transactions, is enforced. As we can see from Figure 5(a), ate of the subclasses increases with increasing load. Admit-All violates the QoS specification as ate^{v,b_v} is greater than the overshoot given by $ate_r^{v,b_v} \times (100 + M_p^{v,b_v})$. Using MCIDS, ate^{v,b_v} reaches its reference ate_r^{v,b_v} and ate^{v,b_v} is less than the overshoot for all subclasses and, hence, satisfying the QoS specification. Note that when ap^{v,b_v} becomes zero then ate^{v,b_v} is equal to zero as we always measure the average transaction error over admitted transactions. Consequently, $ate^{1,1}$ and $ate^{1,2}$ are greater than zero as $ap^{1,1}$ and $ap^{1,2}$ do not reach zero. Turning to Figure 7(a) we see that $mwde$ starts increasing as the load increases, trying to lower the update load so that no user transactions are rejected. Hence, QoD is degraded during high loads in order to accommodate as many transactions as possible.

In summary we have shown that MCIDS provides robust and reliable performance that is consistent with the QoS specification for varying load. The admission mechanism enforces the desired strict hierarchic admission policy, where the least important transactions are rejected and the most important transactions are admitted and executed.

5.5 Experiment 2: Results of Varying esterr

The goal of this experiment is to see how MCIDS reacts to varying execution time estimation error. We show that MCIDS is not significantly affected by the execution time estimations errors. We measure ate, ap, and $mwde$ and vary $esterr$ between -0.2 and 3 with steps of 0.2. This way we examine the effects of both overestimation and underestimation of the execution time. The load is set to 200%. We measure the system performance during steady state when the load is not changing considerably (the transient state performance is examined in detail in Section 5.6). To measure the performance of the system during steady state, we start measuring ate, ap, and $mwde$ after $200s$ to remove the effects of the transients in the beginning of the simulation. Figures 5(b), 6(b), and 7(b) show ate, ap, and $mwde$ when $esterr$ is varied. The dashed lines indicate references, while the dashed-dotted lines give the specified overshoot (according to the QoS specification). The confidence intervals for ate and ap are less than $[ate - 5.09\%, ate + 5.09\%]$ and $[ap - 9.85\%, ap + 9.85\%]$, respectively. The confidence interval for $mwde$ is zero, as $mwde$ is constant for all $esterr$.

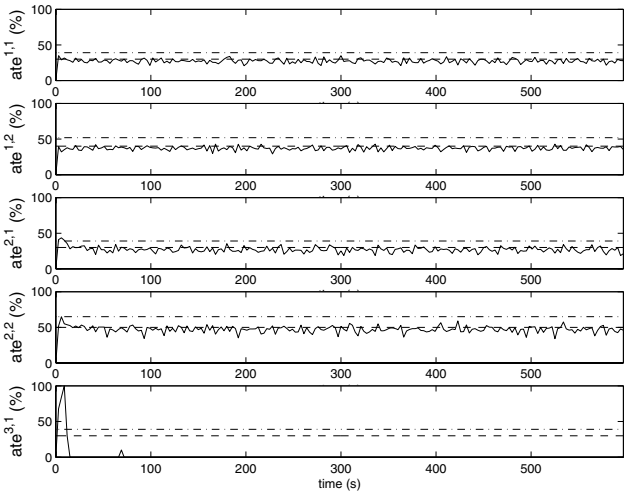


Figure 8. Transient Performance

Ideally, the performance of the MCIDS should not be affected by execution time estimation errors. This corresponds to no or little variations in ate, ap, and $mwde$ as $esterr$ changes. As shown in Figures 5(b), 6(b), and 7(b), ate, ap, and $mwde$ do not change significantly with varying $esterr$. From above we conclude that MCIDS is insensitive to changes to execution time estimation error as ate, ap, and $mwde$ do not change significantly with varying $esterr$. This means that MCIDS conforms to inaccurate execution times, satisfying the QoS specification.

5.6 Experiment 3: Transient Performance

Studying the average performance is often not enough when dealing with dynamic systems. Therefore we study the transient performance of MCIDS by measuring ate. The load is set to 200% and $esterr$ set to zero. Figure 8 shows the transient behavior of ate for all subclasses. The dash-dotted lines indicate specified overshoots (according to the QoS specification), whereas the dashed lines represent references.

The overshoot requirements for the most important subclasses $sbc^{1,1}$ and $sbc^{1,2}$ are satisfied. This also holds for $ate^{2,2}$, which is lower than its overshoot. As we can see $ate^{2,1}$ is somewhat greater than the overshoot, however, the difference between the maximum $ate^{2,1}$ and the overshoot

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

is to small to be considered as a direct violation of the specification. The overshoot of $ate^{3,1}$ is undefined since the steady state value of $ate^{3,1}$ is zero.[3] Hence, the overshoot requirements for the most important service classes are satisfied.

As we can see, $ate^{3,1}$ reaches 100% and it takes a while until $ate^{3,1}$ becomes zero. Using feedback control we are able to react to changes only when the controlled variable has changed and, hence, when $ate^{3,1}$ is greater than $ate^{3,1}_r$, the ATE controller for $sbc^{3,1}$ computes a positive $\delta c^{3,1}_{ATE}$, requesting for more capacity. As the assigned capacity of $server^{3,1}$ is less than the requested capacity ($c^{3,1}$ is near zero), transactions are rejected instead according to the capacity allocation algorithm (see Figure 4). The rejection rate increases as $\delta c^{3,1}_{ATE}$ increases and, hence, a larger $\delta c^{3,1}_{ATE}$ results in improved suppression of $ate^{3,1}$ and faster convergence to zero. The magnitude of $\delta c^{3,1}_{ATE}$ increases as the magnitude of the P-controller parameter K_P (see Section 4.3) increases. Now, we have tuned each controller when the load is 100% and considering that the applied load in the experiment is 200%, the magnitude of K_P of each controller is not sufficiently large to efficiently suppress $ate^{3,1}$.

By increasing the magnitude of the P-controller parameter as the applied load increases, better QoS adaptation is achieved. One way to deal with changing system properties, e.g., the load applied on the system, is to use gain scheduling or adaptive control [23], where the behavior of the controlled system is monitored at run-time and controllers adapted accordingly. In our case, the RTDB reacts to the higher applied load by increasing the magnitude of the P-controller parameter such that faster QoS adaptation is achieved. We believe that using gain scheduling, where the parameter of the P-controllers changes according to the current applied load, results in a substantial performance gain with respect to faster QoS adaptation. In our future work we plan to use gain scheduling to update the control parameters.

6 Related Work

Liu et al. [18] and Hansson et al. [13] presented algorithms for minimizing the total error and total weighted error of a set of tasks. Bestavros and Nagy have presented approaches for managing the performance of RTDBs, where the execution time of the transactions are unknown [4]. Each transaction contributes with a profit when completing successfully. An admission controller is used to maximize the profit of the system. The work by Liu et al., Hansson et al., and Bestavros and Nagy focus on maximizing or minimizing a performance metric (e.g. profit). The latter cannot be applied to our problem, since we want to control a set of performance metrics such that they converge toward a set of references given by a QoS specification. Kuo et al. described the notion of similarity, where transactions that produce similar results are skipped during overloads [14]. However, in the work by Kuo et al. the effects of changes to workload characteristics, such as execution time estimation error, are not reported.

Turning to imprecise data services, the query processor APPROXIMATE produces monotonically improving answers as the allocated computation time increases [24]. The relational database system CASE-DB can produce approximate answers to queries within certain deadlines [20]. Lee et al. studied the performance of real-time transaction processing where updates can be skipped [17]. In contrast to the above mentioned work, we have introduced imprecision at both data object and transaction level and presented QoS in terms of data and transaction imprecision.

Rajkumar et al. presented a QoS model, called Q-RAM, for applications that must satisfy requirements along multiple dimensions such as timeliness and data quality [16]. However, they assume that the amount of resources an application requires is known and accurate, otherwise optimal resource allocation cannot be made. Goddard and Liu, and Brandt et al. presented task models where the parameters of the tasks (e.g., execution time and period) that represent QoS change during run-time [10, 5]. However, their models require that worst case execution times are known, which does not conform to the task model presented in this paper. Further, it is not possible to specify the importance of the tasks in contrast to the model presented in this paper.

Kang et al. used a feedback control scheduling architecture to balance the load of user and update transactions for differentiated real-time data services [15]. However, in this work orthogonality in importance and QoS requirements cannot be realized. In our earlier work we proposed an approach, called RDS, for managing the performance of multi-class real-time data services [3]. However, in that approach only one QoS class is associated with an importance class. In this paper, a generalized approach is presented, where multiple QoS classes are associated with an importance class.

Feedback control scheduling has been receiving special attention in the past few years [19, 21, 2]. However, none of them have addressed QoS management of imprecise real-time data services.

7 Conclusions and Future Work

New emerging applications, e.g., web services and telecommunication, operate in highly unpredictable environments where the workload characteristics cannot be precisely predicted. This makes the schedulability analysis

[3] Recall from Figure 1 that overshoot is defined in relation to the steady-state value of the controlled variable.

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE
IEEE COMPUTER SOCIETY

and overload management very difficult and, consequently, deadline misses may occur or an unacceptable level of QoS may be experienced. As many real-time systems often consist of several applications with varying degrees of importance or criticality, it is vital for a real-time system to consider the importance of the applications and to reject requests from less important applications in the face of an overload. Deadline misses or very low QoS have less impact on low importance applications as opposed to important applications, which may result in a catastrophe. In this paper we present a general approach, called MCIDS, for specifying and managing performance for differentiated and imprecise real-time data services. The performance specification model allows the database operator to specify the importance and the QoS requirement of the transactions. The QoS specification is expressed in terms of desired nominal performance and the worst-case system performance and system adaptability in the face of unexpected failures or load variation. The presented architecture and algorithms enables an RTDB to prioritize more important transactions, ensuring that the most important transactions are processed during overloads. We show through experiments that MCIDS satisfies importance and QoS requirements even for transient overloads and with inaccurate runtime estimates of the transactions.

For our future work we consider other metrics for QoS, e.g., utilization, and adapt techniques from adaptive control [23] to better track changes in the dynamics of the controlled system.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Transactions on Database System*, 17:513–560, 1992.

[2] M. Amirijoo, J. Hansson, S. Gunnarsson, and S. H. Son. Enhancing feedback control scheduling performance by on-line quantification and suppression of measurement disturbance. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[3] M. Amirijoo, J. Hansson, S. H. Son, and S. Gunnarsson. Robust quality management for differentiated imprecise data services. In *Proceedings of the IEEE International Real-Time Systems Symposium (RTSS)*, 2004.

[4] A. Bestavros and S. Nagy. Value-cognizant admission control for RTDB systems. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, pages 230–239, 1996.

[5] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, 2003.

[6] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 1997.

[7] J. Chung and J. W. S. Liu. Algorithms for scheduling periodic jobs to minimize average error. In *Proceedings of the Real-Time Systems Symposium (RTSS)*, 1988.

[8] G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*. Addison-Wesley, third edition, 1998.

[9] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[10] S. Goddard and X. Lin. A variable rate execution model. In *Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS)*, 2004.

[11] T. Gustafsson and J. Hansson. Data management in real-time systems: a case of on-demand updates in vehicle control systems. In *Proceedings of Real-time Applications symposium (RTAS)*, 2004.

[12] J. Hansson, S. H. Son, J. A. Stankovic, and S. F. Andler. Dynamic transaction scheduling and reallocation in overloaded real-time database systems. In *Proceedings of the Conference on Real-time Computing Systems and Applications (RTCSA)*, 1998.

[13] J. Hansson, M. Thuresson, and S. H. Son. Imprecise task scheduling and overload managment using OR-ULD. In *Proceedings of the Conference in Real-Time Computing Systems and Applications (RTCSA)*, 2000.

[14] S.-J. Ho, T.-W. Kuo, and A. K. Mok. Similarity-based load adjustment for real-time data-intensive applications. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS)*, 1997.

[15] K.-D. Kang, S. H. Son, and J. A. Stankovic. Service differentiation in real-time main memory databases. In *Proceedings of the International Symposium on Object-oriented Real-time Distributed Computing*, 2002.

[16] C. Lee, J. Lehoezky, R. Rajkumar, and D. Siewiorek. On quality of service optimization with discrete QoS options. In *Proceedings of the 5th Real-Time Technology and Applications Symposium (RTAS)*, 1999.

[17] V. Lee, K. Lam, S. H. Son, and E. Chan. On transaction processing with partial validation and timestamps ordering in mobile broadcast environments. *IEEE Transactions on Computers*, 51(10):1196–1211, 2002.

[18] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82, Jan 1994.

[19] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son. Feedback control real-time scheduling: Framework, modeling and algorithms. *Real-time Systems*, 23(1/2), July/September 2002.

[20] G. Ozsoyoglu, S. Guruswamy, K. Du, and W.-C. Hou. Time-constrained query processing in CASE-DB. *IEEE Transactions on Knowledge and Data Engineering*, 7(6), 1995.

[21] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance managment. *Real-time Systems*, 23(1/2), July/September 2002.

[22] K. Ramamritham, S. H. Son, and L. C. DiPippo. Real-time databases and data services. *Real-Time Systems*, 28(2-3):179–215, 2004.

[23] K. J. Åström and B. Wittenmark. *Adaptive Control*. Addison-Wesley, second edition, 1995.

[24] S. V. Vrbsky and J. W. S. Liu. APPROXIMATE - a query processor that produces monotonically improving approximate answers. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, December 1993.

Proceedings of the Fourth International Workshop on Web Site Evolution (WSE'02)
0-7695-1804-4/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY