Handling Overload and Data-Relaxation Control in Distributed Real-Time Database Systems

Samia Bouzefrane, Jean-Paul Etienne and Claude Kaiser
Laboratoire CEDRIC, 75141, Paris, France

Abstract

Current applications are distributed in nature and manipulate time-critical databases with firm-deadline transactions. To guarantee performance and availability of such applications fault-tolerant mechanisms need to be developed. In the context of time critical applications, this also leads to providing means to ensure that as many as possible real-time transactions satisfy their deadline in the presence of overload. In this paper, we propose a distributed protocol, which provides mechanisms to handle overload situations while tolerating bounded data inconsistencies in the context of distributed real-time database systems.

Key Words: Real-time database system, overload, real-time transactions, real-time scheduling.

1 Introduction

The rise of technological advances has led to the dawn of wide-spread distributed-oriented applications, such as Web-based services, electronic commerce, mobile systems and multimedia applications, most of which manipulate time-critical databases. Given that these applications are massively accessed in general, maintaining their high-performance and availability is of prime importance. In the case of time-critical applications, this also leads to providing means to ensure that as many as possible real-time transactions satisfy their deadline in the presence of overload. An efficient overload management is crucial in those systems, whereby any mishandling of these situations may entail devastating consequences. One such system is the Emergency Medical Services, exemplified in [18].

In general, timing constraints can be expressed as a direct function of urgently required information, such as the value of a banking account, a stock exchange price or the availability of a product. However, these are also heavily dependent on the maturation of the information system architecture. Indeed, the tightness of the timing constraints of triggered transactions is not only a function of the evolution of functional needs, which may augment the number as well as the nature of requests to be processed by the transactional monitor, but also depends on the topology and heterogeneity of the evolving network infrastructure, as well as the protocol layering issues to ensure data interoperability and security. As a result, both the constant evolving logical and physical environments can increase the latency of triggered transactions, thus reducing the necessary time needed to process the transactions before their deadline expiration.

By and large, with the blending of timing constraints and data transactions, the exact logical consistency that is typical to non real-time databases is difficult to achieve. In fact, real-time constraints may make precise computation impossible. In classical database systems, data imprecision is voided, thanks to the transaction isolation property, which is achieved through transaction serializability. With the latter property, concurrency control of transactions is achieved by managing the execution of overlapping transactions sequentially [11]. However, within real-time databases, serially executing transactions may increase the frequency of deadline misses. To circumvent this problem, some works have attempted to relax this isolation property by tolerating some bounded imprecision in the data being accessed, thus increasing the possibility of transactions being executed concurrently. Indeed, in many applications some imprecision is tolerated. For instance, in multimedia applications, it is acceptable to receive approximate results (some pixels of an image) obtained earlier than accurate ones obtained much later. This is particularly true for massively multiplayer online games. In an internet tennis game [23], based on a client/server architecture, when a player hits the ball, the server uses a kinematics model to determine the ball’s trajectory up to the collision with an object (ground, the raquet of another player). Then it sends to each client (Internet player) the successive positions of the ball at a rate of about thirty times a second. In the dead reckoning approach, this calculation is performed by each client. The server takes care only of the next possible collision. In this case, the server sends the position of the collision as well as the new velocity of the ball as soon as possible, even though these values are not precise, in order to avoid discontinuity in the ball’s trajectory and ensure that its movement remains regular. An unexpected velocity change of the ball might give a bad impression.

While much work has been done in the management of transient overloads in centralized Real-Time Database Systems (RTDBSs) [4-5, 9, 13, and 24], comparatively little has been
done to manage these in a distributed environment. Our contribution lies mainly in the latter case.

In this paper, we focus on the design of a distributed algorithm, based on a commit processing protocol, which provides mechanisms to handle overload situations while tolerating some bounded inconsistencies in the database. Overload occurs when the computation time of a set of transactions exceeds the time available on the site processor, causing the deadline miss. The ε-data concept, initially proposed in [29], is employed here as a correctness criterion to guarantee consistency of the distributed database.

In a RTDBS, transactions are usually classified according to the consequences resulting from a deadline miss [17, 22, 26-28]. Our study is concerned only with “firm-deadline” transactions, that is, transactions whose results are taken into account only if they finish within their deadline. Aborting the transactions does by no means entail dramatic consequences. Firm-deadline transactions are common in wide-spread applications, such as Web-based services, which rely on timeout features of communication protocols to determine successful termination or abortion of requests. As an example, Duvallet, et al. [10] described a system that controls a robot, which collects information on objects moving on a conveyor belt. If the acquisition is not completed before the object is out of sight of the robot, the operation is aborted. It is started over when another object comes in sight again. This operation context is similar to that of on-line services.

The remainder of this paper is organized as follows: Section 2 describes the different mechanisms used by the proposed protocol. Section 3 details the protocol regarding the overload management strategy used and the application of ε-data, a notion close to ε-serializability, to allow more transactions to be executed concurrently. The algorithm of the protocol is presented in Section 4. Section 5 describes the Java platform that implements the protocol. Section 6 presents the different modules and configuration parameters that can be set to evaluate the protocol under various running conditions. In Section 7 we discuss the results of the experiments. We show notably that the protocol exhibits good performance under overload and ε-data conditions. Related works are presented in Section 9 after concluding the paper in Section 8.

2 Principles of the Protocol

The foundation of the protocol lies essentially on two concepts, namely the notion of importance and the ε-data concept. The former concept is used to characterize transactions judged “important” with respect to the application. It is used as a base to discriminate transactions that will have their execution maintained from those that will be rejected during overload situations. This concept consists mainly in assigning a numerical value to transactions to define their degree of importance. Among the techniques that use this concept, Saad, et al. [30] have proposed a protocol to control transactions load in a replicated RTDBS. Here, it is essentially used to handle transient overloads in non-replicated distributed real-time database systems.

The objective of the ε-data concept is to increase transaction concurrency while ensuring logical consistency of the distributed database. As such, besides overhead control, this concept is used as a supplement to increase successful termination of real-time transactions. In general, it is argued that transactions in RTDBS need not have ACID (Atomicity, Consistency, Isolation, Durability) properties, judged too restrictive in a real-time context [27]. To relax these properties, researchers have proposed techniques, based on temporal data semantics, to improve the performance of transaction scheduling [34]. These works have led to the development of forced wait and data similarity policies [20, 33, 12], as well as the epsilon serializability criterion (ESR) [25, 32].

In the next sub-sections, we present the database model used and detail the notion of the ε-data concept [29], as well as the overload management mechanism.

2.1 The Database Model

Our database architecture is decomposed into one master site and several participant sites. The master site is a coordinator process, which handles transactions sent by client entities. It is notably responsible for:

- managing transaction decomposition into sub-transactions,
- coordinating the execution of the sub-transactions by the participant sites, over which the database is distributed,
- informing clients of successful termination or abortion of transactions.

In our model, we restrict ourselves to firm real-time transactions. As a result, successful transaction termination is dependent upon whether or not the result of the transaction is delivered before its deadline miss.

Each participant site manages a cohort process, which handles three modules (see Figure 1):

- a scheduler module, based on the EDF (Earliest Deadline First) scheduling algorithm [21]. EDF ensures that CPU allocation is always guaranteed to the most urgent transaction, while still offering an optimal scheduling for any set of transactions;
- an overload-manager module that controls transient overloads each time a new sub-transaction is inserted in the ready-queue of transactions, and
- a data-manager module that applies ε-data concept upon the presence of conflicting and overlapping sub-transactions at the participant site.

Our database model is distributed over participant sites, with no consideration for data replication. In such case, data elements are scattered over participant sites rather than duplicated. We proposed an overload management solution under such setup in [30]. As regards to global transaction structuring, we consider transactions comprising sub-transactions, having each several query or update operations. However, within a sub-transaction, operations are applied to
distinct data, that is, there will be no two operations pertaining to the same data record. This ensures that data conflicts occur only among sub-transitions and not within sub-transactions.

**2.2 $\varepsilon$-Data Notion**

In RTDBSs that control emergencies, partial or approximate results obtained on time may be more useful than precise or complete results obtained too late. This is the case of multimedia applications, whereby losing a few packets (composing audio or video streams) in the favor of performance is tolerated. However, this will not lead to serious consequences, only audio or video quality will be reduced.

The epsilon-serializability [25, 32] is a generalization of traditional serializability, which tolerates bounded inconsistency in transactional processing. Concretely, an update transaction may export certain inconsistencies when it updates a data item that is being accessed by read transactions. On the other hand, a read transaction may import certain inconsistencies when it reads a data item that is being updated. Consequently, both update and read transactions may handle inconsistent data.

The concept of $\varepsilon$-data follows this line of reasoning by favoring punctuality instead of data consistency. It is essentially applied to read transactions, which conflict and overlap with update transactions during their execution.

The $\varepsilon$-data concept [29] deals with absolute data value imprecision, which is a percentage of the current data value. For example, let a data item $d = 20$. If $d$ tolerates an imprecision $\varepsilon$ then it is called $\varepsilon$-data. Let $\varepsilon$ be equal to 10 percent of $d$ value ($\varepsilon = 0.2$), then we tolerate the use of values being in the interval $[d-\varepsilon, d+\varepsilon] = [20-2, 20+2]$. We denote by $\varepsilon$-data, a data item whose exact value is $v$ and where each value in $[v-\varepsilon, v+\varepsilon]$ is acceptable. Generally, a data item $d$ is associated to two values, an $afterValue$ and a $beforeValue$. The system saves the $beforeValue$ for the lifespan of the sub-transaction that updates the data item, such that this value can be restored if the sub-transaction is aborted. Moreover, if the $afterValue$ is in the $[beforeValue - \varepsilon, beforeValue + \varepsilon]$ interval, then all the read sub-transactions that are executed before the validation of the update sub-transaction, can use the $beforeValue$. A greater number of sub-transactions can then be carried out in parallel, thus increasing their chance to meet their deadline. This consequence is particularly important in financial applications, such as banking, stock exchange and insurances domains, whereby 70 percent of transaction processing time is spent in handling query transactions. In general, considering either data freshness (temporal precision) or value approximation (functional precision), or even a mixture of both, will depend essentially on the nature of the application under study. For instance, in the numerical analysis domain, the $\varepsilon$-data concept enables both the expression of data freshness as well as the estimation of the evolving data. We can calculate $afterValue$ as a smoothing of the last observed values, while deciding whether or not to use $\varepsilon$. However, when dealing with alphabetic or graphic data items, it would be necessary to define the notion of distance, which is more complex. This is out of scope of this paper.

By and large, in a strict context, if a query transaction requests a lock on a data item $d$, which is already locked by an update transaction, then it will either wait or abort. Conversely in $\varepsilon$-data applications, this isolation property is relaxed. A read transaction can access concurrently a data item being updated by a write transaction, only if the amount of inconsistency introduced by the latter is less than a specified bound $\varepsilon$. In this latter case, it is obvious that the lock management policy used will differ from the one used in classical RTDBSs.

**2.3 Overload Management**

In general, each submission of a new sub-transaction to a site may potentially cause an overload, whereby the aggregate computation time of the set of ready sub-transactions exceeds the computing capability of the site processor to fulfill their deadline. Even though the EDF scheduling algorithm guarantees an optimal scheduling of transactions under normal situations, however, this is not the case during overloads. Under such conditions, EDF becomes overly aggressive in aborting transactions and does not apply a deterministic rejection strategy. To resolve this problem, EDF is supplemented with an overload-management policy. Here, the aim is to favor the execution of the most important transactions of the application. To this end, transactions are classified according to a numerical parameter called importance value [30]. Upon the arrival of a new sub-transaction, the overload manager determines whether the latter might cause an overload. As soon as an overload situation is detected, the rejection strategy bases itself on the importance of transactions to determine the order in which they will be aborted. In general, rejected transactions are those having the lowest numerical importance value. Hence, this guarantees determinism of the rejection policy.
2.3.1 Transaction Importance. In our model, each transaction is characterized by a deadline, which defines its urgency and an importance value, which defines the criticality of its execution. The importance value can be assigned by the user or the application designer. In general, the bigger the importance value of a transaction, the more essential it is for the application. Moreover, the importance and deadline of a transaction may not be directly related. Hence, if two different transactions have the same deadline, they do not necessarily have the same importance values. The importance values can be assigned according to classes of transactions, with each class corresponding to a particular application domain. Let us consider some application examples [3]. In the banking sector, bank card authorization operations (essentially read operations with a response time lower than 300ms) are more significant than ATM operations (having a response time lower than 300ms, with 70 percent of the operations being read operations), which in turn are more important than remote operations (having a response time ranging from 300ms to 400ms, with 80 percent of the operations being read operations). In the Stock Exchange domain, Stock Exchange orders (having a response time lower than 300ms, with 70 percent of the operations being read operations) are more important than operations carried out on a Web portal (essentially read operations, having a response time ranging from 300ms to 400ms). In the insurance domain, cash applications (such as profile contract, subscription contract, etc.), having a response time ranging from 400ms to 500ms, with 70 percent of the operations being read operations, are more important than back-office applications, which manage contracts (validation, cancellation, green cards edition, etc.), having 70 percent of the operations being update operations.

In our model, a global transaction is characterized by an arrival time, a deadline, and an importance value. Likewise, at a participant site, a sub-transaction is characterized by an arrival time, an execution time, a deadline and an importance value. It should be noted that the deadline and importance value of a sub-transaction are inherited from the global transaction to which it belongs. In general, the deadline of a transaction must be higher than $2 \times \delta$, with $\delta$ being the communication transmission time. This gives the assurance that deadline expiration of sub-transactions will not occur before the activation of the sub-transactions within participant sites or before obtaining an answer concerning their successful termination or abortion. Xiong and Ramamritham [35] and Ramamritham, et al. [28] present in much detail how deadlines and periods of real-time transactions are calculated according to the temporal validity of the data being manipulated.

2.3.2 The Stabilization. The stabilization mechanism handles system overload by maintaining in the ready queue only the sub-transactions that will meet their deadline. As stated previously, the strategy is to favor the execution of sub-transactions having high importance values when an overload is detected within a site. Thus, upon an overload, sub-transactions with lowest importance values are dropped from the ready queue and are therefore aborted until the processor laxity recovers a positive value. The processor laxity, at time $t$, is the maximum time the processor may remain idle, after $t$, without causing a transaction to miss its deadline. In the case where sub-transactions have identical importance values, the less urgent sub-transactions are dropped first.

3 Protocol Description

Our protocol is designed to manage distributed real-time transactions. It focuses on firm real-time transactions and uses the model described in Section 2. This protocol detects and handles temporal overloads within each participant site while providing means to increase transaction concurrency, namely through the use of the $\epsilon$-data concept. To handle communication between sites, our protocol uses the same mechanisms as the 2PC (two Phase Commit) protocol, presented in [31] and [2].

3.1 Integrating Overload Control

Our protocol leverages the 2PC protocol to handle distributed transaction processing under overload situations. When the coordinator process receives a transaction request $\tau$, it decomposes the latter into sub-transactions in order to send them to corresponding cohort processes for execution. As such, for each sub-transaction $T_i$, the coordinator sends an \textsc{Initiate($T_i$)} message to execute $T_i$ on the cohort process that manages data items needed by $T_i$. When a cohort receives the \textsc{Initiate($T_i$)} message, it applies the stabilization process to determine whether the arrival of $T_i$ might cause an overload. Upon such a situation, the ready queue of transactions is stabilized by dropping sub-transactions having the lowest importance values. Once a sub-transaction is dropped, the cohort informs the coordinator of the abortion by sending a “NO” vote. In turn, upon reception of the vote, the coordinator broadcasts ABORT messages to all the cohorts for them to invalidate their corresponding local sub-transactions. However, if the coordinator receives YES messages from all its cohorts, it concludes that all the sub-transactions of $\tau$ have been executed successfully within the respective participant sites. In this case, the coordinator will broadcast COMMIT messages to the latter, provided that the messages can be received before deadline expiration of the global transaction. In the case of deadline expiration, the coordinator does not send a response to its cohorts as either the coordinator itself or the cohorts will detect it locally. In general, deadline expiration of a transaction entails abortion of the latter.

Before presenting how the integration of the $\epsilon$-data concept within each cohort increases sub-transactions concurrency, a formalization of the stabilization process is presented in the next section.

3.2 The Stabilization Process Description

A sub-transaction $T_i$ within a participant site $s$ is characterized by four temporal parameters: $r_i$, its arrival time at $s$, $C_i$, its computation time, $D_i$, its absolute deadline inherited
from the global transaction \( \tau \) to which it belongs and \( \text{Imp}_i \) its importance value. Following these parameters, \( D_r - r_i \) denotes the relative deadline of \( T_i \) and \( C_i(t) \) represents the remaining computation time of \( T_i \) at time \( t \).

Let \( \text{readyQueue}_{s,t} \) be the list of ready sub-transactions at participant site \( s \), sorted at time \( t \) according to the EDF policy:

\[
\text{readyQueue}_{s,t} = \{ T_0, T_1, ..., T_n \} \quad (\forall i \in \{ 1, n \})
\]

The conditional laxity \( CL_i(t) \) of sub-transaction \( T_i \) within \( \text{readyQueue}_{s,t} \) represents the highest time interval during which \( T_i \) may be delayed without missing its deadline. It is formalized as follows:

\[
LC_i(t) = D_i - t - \sum_{j=0}^{i} C_j(t)
\]

Here the result takes into account the pending execution time of all sub-transactions having earlier deadline than \( T_i \). From this definition, the processor laxity \( PL \) for a site \( s \), at time \( t \), is then equal to the conditional laxity of the sub-transaction having the lowest conditional laxity within \( \text{readyQueue}_{s,t} \). Hence:

\[
PL(t) = \text{laxity}(\text{readyQueue}_{s,t}) = \text{Min} \{ CL_i(t) \} \quad (\forall T_i \in \text{readyQueue}_{s,t})
\]

In general, an overload situation is detected as soon as the site laxity \( PL(t) \) is less than 0, with late sub-transactions being those having a negative conditional laxity. The overload value is then equal to the absolute value of the processor laxity, \( |LP(t)| \). Hence:

- If \( LP(t) > 0 \Rightarrow D_i > t + \sum_{j=0}^{i} C_j(t) \) then all \( T_i \) will finish its execution before expiration of its deadline, while allowing the execution of \( (i-1) \) sub-transactions preceding it in the queue.
- If \( LP(t) < 0 \Rightarrow (\exists T_k) D_k < t + \sum_{j=0}^{k} C_j(t) \), then there exists at least one ready sub-transaction \( T_k \) for which the deadline expires before the end of its execution. This is exemplified in Figure 2, whereby \( T_3 \) will generate an overload at time 8. To ensure that the ready queue will contain only sub-transactions meeting their deadline, we have to stabilize the queue by determining which sub-transactions to abort each time an overload is detected. We formalize the notion of stabilization as follows:

We say that \( \text{readyQueue}_{s,t} \) is a stabilization of \( \text{readyQueue}_{s,t} \) if:

1) \( \text{readyQueue}_{s,t} \subseteq \text{readyQueue}_{s,t} \),
2) \( \text{laxity}(\text{readyQueue}_{s,t}) \) \( \geq 0 \) and
3) if \( \forall \mathcal{R} = \text{readyQueue}_{s,t} - \text{readyQueue}_{s,t} + \text{readyQueue}_{s,t} \) then \( (\forall T_i \in \mathcal{R}) (\forall \\kappa \subseteq \text{readyQueue}_{s,t} \} \) we have:

\[
\forall T_i \in \mathcal{R}, \text{Imp}_i < \text{Imp}_{\kappa} \Rightarrow \text{laxity}(\text{readyQueue}_{s,t} - \kappa \cup \{ T_i \}) < 0.
\]

When an overload is detected within a participant site, due to the arrival of a new sub-transaction \( T \), the stabilization process removes from \( \text{readyQueue}_{s,t} \) a sub-transaction \( T' \) having the lowest importance value and which removal contributes to augment the conditional laxity of \( T \). This leads to choose \( T' \) from the set of sub-transactions that are more urgent than \( T \). However, if there is no such sub-transaction, then \( T \) itself is removed from \( \text{readyQueue}_{s,t} \). The operation is repeated until the processor laxity recovers a positive value. This is expressed in point 2. Point 3 is explained using the following example. Let us consider three sub-transactions \( T_1, T_2 \) and

\[
C_i(T_1)=2 \quad D_j=5 \\
C_i(T_2)=3 \quad D_j=7 \\
C_i(T_3)=4 \quad D_j=8
\]

\[
LP = \text{Min}(LC_{T_1}, LC_{T_2}, LC_{T_3}) = -1
\]

Overload value = \( |LP| = 1 \)

![Figure 2: Processor-laxity example](image)
$T_3$ sorted by increasing importance values (with $T_3$ being more important than $T_2$, being more important than $T_1$). Upon the detection of an overload, suppose that dropping $T_1$ does not resolve the situation, whereas aborting $T_2$ alone would. As a result, dropping $T_1$ becomes useless. In this case, $T_1$ is kept in the system.

3.3 Integrating ε-Data Concept in the Locking Condition

As presented in Section 2, the ε-data concept allows us to increase transaction concurrency within a participant site by relaxing the database isolation property. As such, it affects how a locking condition is handled within the database. Particularly, such a condition is incurred whenever a query sub-transaction $Q^e_i$ tries to read-lock a data item $d$, which is already write-locked by an update sub-transaction $W^e_i$. In our case, instead of blocking or aborting $Q^e_i$ as in classical protocols, the latter is allowed to pursue concurrently its execution with $W^e_i$, provided that the difference between the value written by $W^e_i$ and the value read by $Q^e_i$ does not exceed the threshold ε. However, in the case where the value written by $W^e_i$ exceeds this threshold, then the transaction manager behaves classically. In general, the ε-data concept operates at two levels:

- During the execution phase of an update sub-transaction: here, all read sub-transactions requesting a data item, which is write-locked by an update sub-transaction that is in its execution phase, may execute in parallel with the latter, provided that the amount of inconsistency it has introduced is bounded by ε.

- During the uncertainty phase of an update sub-transaction: the uncertainty phase of a sub-transaction begins at the time it finishes its execution and starts waiting for a COMMIT or an ABORT message. However, this phase may last for a significant amount of time due to varying communication time delays between sites. Here also, if the amount of inconsistency induced by the update sub-transaction is within the threshold range, then all read sub-transactions requesting the data item will be allowed to access its original value and therefore be allowed to continue their execution in parallel with the update sub-transaction. In general, during the uncertainty phase, read locks are released while write locks are kept until the sub-transaction validation.

4 Algorithm

Our protocol provides basically three functionalities:

1) It handles serializability of the transactions. As presented in Section 3, this is achieved via a two phase commit protocol augmented with mechanisms to handle real-time features.

2) It conducts admission control of new sub-transactions and handles overload situations by applying a stabilization routine, which takes into account importance values of sub-transactions. This is achieved by each cohort process.

3) It handles transaction concurrency and consistency at each participant site by applying the ε-data concept and by blocking or aborting conflicting sub-transactions.

4.1 Notations

- $\tau$ denotes a global transaction submitted to the master site. It is composed of $k$ sub-transactions denoted by $\{T_1, T_2, ..., T_k\}$, which will be executed on $k$ participant sites.

- $Imp$ is a positive integer value that denotes the importance value of $\tau$. The importance values of sub-transactions are equal to the importance value of the global transaction to which they belong. Hence, $Imp = Imp_i$ for each $i$ in $\{1, k\}$.

- $D$ denotes the absolute deadline of $\tau$. $D = D_i$ is also the deadline of the sub-transactions of $\tau$.

- $readyQueue_s$ is a list that contains ready sub-transactions within a site $s$. The sub-transactions are sorted by increasing absolute deadlines. Thus, the sub-transaction with the highest-priority is the one with the nearest deadline.

- $importanceQueue_s$ contains ready sub-transactions of a site $s$, sorted by increasing importance values. Hence, the first sub-transaction of the queue is the least important one.

- For each data item $d$, we define an ε value, which represents a percentage of its current value. Moreover, $d$ is also associated to two values, an after-value and a before-value. The system remembers the before-value until the successful termination or abortion of the sub-transaction performing an update operation on the data item. It is used to restore the current value of the data item in the case of an abortion. The after-value is used during the application of the ε-data concept. If it is in the range $[before-value - \varepsilon, before-value + \varepsilon]$, then all query sub-transactions arriving before the commitment of the update sub-transaction may proceed concurrently using the before-value, thus increasing their chance to meet their deadlines.

4.2 Exchanged Messages

- $INITIATE(T_i, \tau, D, Imp)$ is a message sent by the master to the participant site containing the data items needed by $T_i$.

- $YES(T_i, \tau)$ is a message sent by a participant site to the master when $T_i$ enters its uncertainty phase (i.e., $T_i$ has finished its execution before its deadline and is waiting for the decision of the master).

- $NO(T_i, \tau)$ is sent by the participant site to the master whenever $T_i$ is aborted before expiration of its deadline. In this case, the abortion of $T_i$ may be the consequence of an overload within the participant site.

- $COMMIT(T_i, \tau)$ is sent by the master to the participant site in order to commit $T_i$.

- $ABORT(T_i, \tau)$ is sent by the master to the participant site in order to abort $T_i$. 
4.3 Text of the Algorithm

The following algorithms describe the behavior of the master, as well as that of each participant site.

5 The Java Simulation Platform

The simulation platform has been developed using Java technologies and handles MySQL databases. The architecture is composed of a master site and three participant sites over which the database system is distributed. The master site handles transaction requests via HTTP. Upon each request, it decomposes the latter into sub-transactions and distributes them to the corresponding participant sites, where they are processed into SQL statements and executed. In general, transactions requests are composed solely of query and update operations, with sufficient conflicts to guarantee the database operations are executed in mutual exclusion, locked database using mutual exclusion mechanisms.

The master site is basically a Java Servlet, which operates on a TomCat Server. To ensure simultaneously handling of transactions, each request sent to the master site is handled by an instance of the Servlet. In general, a transaction request contains one or more data operations, with each operation including the name of the record on which the operation will be carried out, the operation type (read/write), the value of the record (in the case of a write operation) and information pertaining to the participant site, which handles the record.

Each participant site handles a certain number of queues. One of them is the UncertainQueue, which contains completed transaction elements waiting for the commit/abort decision of the master site. The participant site is composed of several thread-based modules implementing the Overload Manager, the EDF Scheduler and the Data Manager, each of which is accessing concurrently the different queues, as well as the local database using mutual exclusion mechanisms.

In our platform, lock handling is controlled by two tables, finaldata and tmpdata. Permanent records are stored in finaldata, while temporary ones are stored in tmpdata. Since database operations are executed in mutual exclusion, locked data items are determined during the application of the ε-data concept by querying tmpdata. This table contains information pertaining to the participant site, which handles the record.

<table>
<thead>
<tr>
<th>Master Site</th>
<th>Participant Site</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>When</strong> a client submits a transaction τ to the coordinator process:</td>
<td><strong>Upon reception</strong> of the message ( \text{INITIATE} (T_i, \tau, D, \text{Imp}) ):</td>
</tr>
<tr>
<td>/ * τ est splitted into ( k ) sub-transactions */</td>
<td>do</td>
</tr>
<tr>
<td>( \forall i \in {1, k} ) :</td>
<td>compute the processor laxity;</td>
</tr>
<tr>
<td>send ( \text{INITIATE} (T_i, \tau, D, \text{Imp}) ) to the site ( s_i ) <em>/ ( s_i ) contains the data required by ( T_i )</em>/</td>
<td>If the laxity is negative Then</td>
</tr>
<tr>
<td></td>
<td>Stabilize ( \text{readyQueue} );</td>
</tr>
<tr>
<td></td>
<td>If a sub-transaction ( T_i ) is cancelled</td>
</tr>
<tr>
<td></td>
<td>Then</td>
</tr>
<tr>
<td></td>
<td>send message ( \text{NO}(T_i, \tau') ) to the master site;</td>
</tr>
<tr>
<td><strong>If</strong> ( k ) messages of type ( \text{YES}(T, \tau) ) are received</td>
<td><strong>EndIf</strong> <strong>EndIf</strong></td>
</tr>
<tr>
<td><strong>Then</strong></td>
<td><strong>Done</strong></td>
</tr>
<tr>
<td>( \forall i \in {1, k} ) send ( \text{COMMIT}(T_i, \tau) ) to the site ( s_i );</td>
<td>If ( (T_i ) has finished its execution before its deadline) Then</td>
</tr>
<tr>
<td><strong>Finsi</strong></td>
<td>send ( \text{YES}(T_i, \tau) ) to the master site;</td>
</tr>
<tr>
<td></td>
<td>( T_i ) enters in an uncertainty phase;</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>send ( \text{NO}(T_i, \tau) ) to the master site;</td>
</tr>
<tr>
<td></td>
<td><strong>endIf</strong></td>
</tr>
<tr>
<td><strong>Upon reception</strong> of a message ( \text{NO}(T, \tau) )</td>
<td>If a message ( \text{COMMIT}(T_i, \tau) ) is received</td>
</tr>
<tr>
<td>do</td>
<td>then Commit ( T_i );</td>
</tr>
<tr>
<td>( \forall i \in {1, k} ) send ( \text{ABORT}(T_i, \tau) ) to the site ( s_i );</td>
<td><strong>endIf</strong></td>
</tr>
<tr>
<td><strong>Done</strong></td>
<td>If a message ( \text{ABORT}(T_i, \tau) ) is received</td>
</tr>
<tr>
<td></td>
<td><strong>Then</strong> Abort ( T_i );</td>
</tr>
<tr>
<td></td>
<td><strong>endIf</strong></td>
</tr>
<tr>
<td>If ( (T_m ) is a read sub-transaction that wants to access to data item ( d ) that is already witleocked) Then</td>
<td>If ( (\text{afterValue}(d) \in [\text{beforeValue}(d) - \varepsilon(d), \text{beforeValue}(d) + \varepsilon(d))] )</td>
</tr>
<tr>
<td></td>
<td><strong>Then</strong> */ ε-data */</td>
</tr>
<tr>
<td></td>
<td>Pursue the execution of ( T_m );</td>
</tr>
<tr>
<td></td>
<td><strong>Else</strong> according to the deadline, ( T_m ) is blocked or aborted. In this last case, send ( \text{NO}(T_m, \tau') );</td>
</tr>
<tr>
<td></td>
<td><strong>endIf</strong></td>
</tr>
<tr>
<td></td>
<td><strong>endIf</strong></td>
</tr>
</tbody>
</table>
about write-locked data items held by sub-transactions during their execution. In general, these record elements will remain within the table as long as the sub-transactions holding the data items are not committed or aborted. Upon successful termination of an update sub-transaction, information pertaining to its write-locked data in the tmpdata table is used to update the finaldata table before deletion. In the case where a query sub-transaction performs a read operation on a data record having no write-lock, then operation is performed directly on the finaldata table, otherwise the execution is performed on the tmpdata table, provided that the after-value of the data item lies within an acceptable $\varepsilon$-data range.

6 Managing an Experiment

The application developed to evaluate the performance of our simulation platform is composed of three modules:

- **Execution Module**,  
- **Configuration Module** and  
- **Statistics Module**.

6.1 The Execution Module

The **Execution** module is responsible for sending simultaneous transaction requests via HTTP to the master site. Every request, is composed of three sub-transactions, each of which contains at most three data items randomly chosen from the corresponding participant sites. The results transmitted back by the master site are saved in a database for future analysis. Each result contains the following information: the transaction number, its computation time, its importance value, as well as the result of its execution (commit/abort).

6.2 The Configuration Module

The **Configuration** module enables us to change the execution parameters of the distributed system (see Figure 4), presented below:

- **ReadTime**: defines the execution time of a read operation (700ms, Figure 4),  
- **WriteTime**: defines the execution time of a write operation (800ms, Figure 4),  
- **AddTime**: defines the additional time that is added to the transaction execution time to determine its deadline. An overloaded situation may be generated by adjusting this time,  
- **ImpRead**: defines the importance value of a transaction composed of read operations only (3, Figure 4),  
- **ImpWrite**: defines the importance value of a transaction composed of write operations only (5, Figure 4),  
- **ImpReadWrite**: defines the importance value of a transaction composed of both read and write operations (4, Figure 4),  
- **Epsilon**: defines the degree of imprecision that a data record can tolerate and
- **ConsiderImp**: determines whether the system should cater for the importance value of the transactions during stabilization.

All these parameters enable us to determine the conditions under which the transactions would be executed. Hence, we can determine whether the ε-data concept should be applied during database access or whether the importance value of transactions should be considered during stabilization. The frequency of overload situations can also be adjusted by manipulating the *ReadTime*, *WriteTime* and *AddTime* parameters. The importance value of the different type of transactions can also be modified through the *ImpRead*, *ImpWrite* and *ImpReadWrite* parameters.

In general, the absolute deadline of a transaction is calculated as the addition of its computing time to the *AddTime* parameter, with the computing time depending on the number of read (*nb_read_op*) and write (*nb_write_op*) operations:

\[
\text{Absolute deadline} = \text{Computing time} + \text{AddTime}
\]

\[
\text{Computing time} = \text{nb_read_op} \times \text{ReadTime} + \text{nb_write_op} \times \text{WriteTime}.
\]

### 6.3 The Statistics Module

The purpose of the *Statistics* module is to analyze and display the series of execution grouped according to the type of configuration under which they have been executed. So far, two types of analysis can be performed:

- Percentage of transactions meeting their deadline, grouped by series of execution (Figure 5).
- Percentage of transactions meeting their deadline, grouped by their importance value (Figure 6).

### 7 The Experiments Carried Out

#### 7.1 The Experimental Plan

The experiments that we have been realized using the simulation platform have the primary objective of demonstrating the well-foundness of our protocol regarding the handling of overload situations, as well as the handling of transaction concurrency and consistency. More precisely, the idea is to evaluate the ability of the stabilization process (which relies on the notion of importance) to control efficiently overload situations, as well as the capacity of the ε-data concept to maximize the number of transactions meeting their deadline while guaranteeing database consistency. In what follows, we recall the assumptions made and that we wish to confirm through these experiments. In general, tolerating data imprecision according to the ε-data concept makes sense only in the case where read transactions are trying to access data items locked by concomitant write transactions. In order to ensure the happening of such situations, it is necessary to have
overlapping and conflicting read and write transactions running in the system. During our experiments, such conditions are created by initiating short read transactions of high priority, during the execution of long write transactions with low priorities. Since the scheduling policy of transactions is based on EDF, read transactions are given higher priority by assigning them closer deadlines. The importance concept is taken into account only during transient overloads. To better discriminate transactions upon an overload, each transaction is assigned a distinct importance value.

Our experimental plan is presented as follows:

The first experiment is carried out on a series of conflicting query and update transactions. However, to enable a better appraisal of the results, few conflicting write operations are considered. Here, write transactions are long and have large absolute deadlines, while read transactions are short and have small absolute deadlines. Read/write transactions have intermediate behaviors.

The purpose of this first experimentation is primarily to determine the proper calibration of the configuration parameters and profiles (reading, writing or read/write) of the generated transactions, which will guarantee a high concurrency rate of transactions. The experimentation is carried out first with the ε-data concept and the notion of importance voided in order to enable the assessment of the percentage of the transactions meeting their deadline only under EDF scheduling. Thereafter it is carried out with an imprecision threshold of 15 percent, but still with the notion of importance voided.

In the second experimentation, the objective is to favor the execution of read transactions, notably by giving them greater importance values. It is carried out based on the assumptions verified by the preceding experimentation. Here, to ensure such behavior, each transaction is assigned an importance value, which is inversely proportional to its relative deadline. For this experiment, we expect the concurrency rate of transactions to be higher, with an even better gain when data-imprecision is tolerated. We should also observe that the number of aborting read transactions is kept low.

Compared to the previous experiment, the objective of the third one is to favor the execution of write transactions, i.e., long transactions, by assigning to each transaction an importance value, which is proportional to its relative deadline. For this experiment, we expect the rate of aborting read transactions to be high, with a decreasing concurrency rate of transactions (because of the long transactions). Hence, it is assumed that this concurrency rate will be lowered, with an insignificant gain when data-imprecision is tolerated. Moreover, we should also observe that the number of aborting write transactions is kept low under such conditions.

The objective of the fourth experimentation is to show that in the case where read and write transactions are not favored, the protocol cannot influence the concurrency rate of transactions, and this, even if data-imprecision is tolerated. To verify this assumption, each transaction is assigned a non significant importance value, which is calculated as the absolute difference between the mean relative deadline of all transactions and the relative deadline of the transaction. Here, we expect that, as several transactions may have identical importance values, the abortions will have a more scattered effect on the execution of transactions. Under such conditions, we should find a situation similar to the one where the notion of importance is not used.

7.2 Tuning the Configurations

In all the experiments carried out, several series of execution have been used to measure the performance of the system under different working conditions. The series vary in terms of the number of transactions (50, 100, 150, 200, 250 and 300) in order to evaluate the behavior of each simulation configuration vis-à-vis a linear increase of the number of requests. Every 100ms, a new transaction is submitted to the master site. The series have been designed in a way to ensure several conflicting data access between read and write operations and few conflicts between write operations. Besides, the modifications brought by write operations have an ε-imprecision threshold of 15 percent for a better appraisal of the performance of the ε-data concept during analysis. All the tests have been carried out on a platform consisting of three participant sites, each of which have a local database of 30 records. Also, each distributed transaction contains three sub-transactions, one for each participant site, having each at most three data operations.

During the experimentation, to ensure that each test does not take advantage of the modifications brought by previous ones, the database records are reset to their initial value at the end of each series. Moreover, since transactions are sent concurrently to the master site, their order of arrival as well as their execution may differ from one series to another. To this end, the series are executed several times under each configuration, with the final result being a mean computation of the execution results.

7.3 Results

In each case, each execution series is launched twice to evaluate:

- The influence of the importance when the ε-data concept is not applied,
- The efficiency of the ε-data, in terms of concurrent-reading rate during writing.

The results obtained are gathered in Tables 1 and 2 and Figures 5 and 6.

7.3.1 Policy 1: the importance is not used, only the concept of ε-data is applied. Here, the results obtained (see curves 7 and 8 of Figure 6) show that by applying the ε-data concept alone we get an average increase of 3.72 percent in the number of transactions successfully terminating their execu-
With the importance parameter used to favor read transactions. Under such policy, each transaction has an importance value, which is inversely proportional to its computation time. As stated previously, transactions, which comprise only read operations are by construction shorter than write and read/write transactions. The shortest one has an importance value of 20 while the longest one has a value of 6. The discriminating characteristic of the importance value over the success rate is given in Figure 6.

Curves 1, 2 and 8 of Figure 5 demonstrate that the importance policy has a very positive effect on the number of transactions successfully terminating their execution, with an average increase rate of 9.14 percent, when compared to the case where this policy is voided. We can also notice that when this policy is coupled with the ε-data concept, we get an additional average success rate of 1.91 percent. The experimentation also shows that on 861 read accesses, 90 meet a lock and 61 are accepted, which gives a concurrency rate of 10.5 percent and an ε-data efficiency of 68 percent.

7.3.3 Policy 3: the importance parameter is used to favor write transactions. Under such policy, the importance value of each transaction is proportional to its duration (write transactions are the longest) and is further increased by a value that still favors write transactions.

Curves 3, 4 and 8 of Figure 5 show in this case that the percentage of transactions terminating successfully their execution is quite low. Indeed, under overload conditions, the system is stabilized by eliminating short transactions, which is composed essentially of read operations. In this case, to recover the laxity of late and long transactions, the stabilization process needs to abort a lot of low-laxity transactions (i.e., read transactions). The result obtained here is worse than the one obtained when importance is not used, whereby fewer long transactions are kept and fewer short transactions are removed. As we can see, this is characterized by a considerable decrease of 5.71 percent in the average success rate. Under such policy, even the application of the ε-data concept does not resolve the situation. It improves the success rate by only 0.19 percent, as with the abortion of a great number of read transactions, the concurrency rate of transactions is only 6 percent and hence, the potential use of the ε-data concept is also affected (its efficiency is only of 56 percent).

7.3.4 Policy 4: the importance parameter is used to favor transactions with mean execution time. Under such policy, each transaction has an importance value, which is calculated as the absolute difference between the mean relative deadline of all transactions and the relative deadline of the transaction. In this case, we observe (see curves 5, 6 and 8 of Figure 5) a light improvement in the percentage of transactions terminating successfully their execution when comparing the result to the one obtained when importance is not used (an average increase of 3.62 percent). Here, we can also notice that, when coupling this policy with the ε-data concept, we get an additional average success rate of 2.86 percent. On 864 read accesses, 66 meet a lock and 32 are accepted, which gives a concurrency rate of 8 percent and an ε-data efficiency of 48 percent.

7.4 Assessment of the Results Obtained

When comparing the simulated policies, we can notice that in general, the combination of the notion of importance with the ε-data concept enhance the performance of the protocol under transient overloads. This improvement is further amplified when the importance criterion is configured to favor read transactions. Under such policy, the contribution of the ε-data concept offers a significant increase in the amount concurrency between read and write operations. Figure 4 presents the percentage of transactions meeting their deadline for each series under the four policies described above.

Obviously, these interpretations are related to the initial choices made when calculating the deadlines and importance values of transactions. These configurations may reflect choices that can be imposed by a particular real-time application. For instance, an application may need to privilege speed of read operations to the detriment of the freshness of stored information, when the real-time database is used only for reading and updating data records (for example, directories). We have seen that, in this case, the ε-data concept complements the effect of the importance. However, in an
application, where write operations need to be processed before strict deadlines (for example, financial quotations), the configurations may instead affect negatively overload handling.

8 Conclusion

In this paper, we presented a distributed algorithm based on a commit processing protocol, which handles transient overloads while tolerating some bounded inconsistencies in the database. When an overload is detected within a participant site, transactions that are judged important have their execution maintained while others are discarded until the system is stabilized again. The $\varepsilon$-data concept is employed to increase transaction concurrency while ensuring consistency of the distributed database. To demonstrate the well-foundness of our protocol, we have built a simulation platform, based on Java technology. The master is implemented as a Java Servlet and operates on a Tomcat server. Each participant site implements an overload manager, an EDF scheduler and a data manager. The platform also integrates a graphical interface to submit transactions, a configuration module to specify a certain number of parameters and a statistical module that displays the simulation tests in a graphical way. The experimental results show that the complementary roles of the $\varepsilon$-data and the importance concepts enhance the performance of the protocol under transient overloads.

Our work can be used as a basis for enhancing transactional monitors with mechanisms, which will allow them to handle efficiently requests constrained by temporal constraints and this even in overload situations. Thus, to limit the rate of read transactions missing their deadline, we can give some advice to the developers, like “making short transactions”, “updating a transaction as late as possible”, and “privileging the readings without integrity” [3].

9 Related Work

As stated by [16], “scheduling focuses on when to execute transactions, and overload management focuses on selecting which transactions should be allowed to execute”. In general, there are basically two solutions to the overload problem. One trivial solution is to add more resources to the system. Another solution consists in resolving the problem through transaction termination or rejection. With the “transaction termination” policy, overloads are resolved by aborting transactions eligible for execution, whereas “transaction rejection” implies rejecting arriving transactions causing an overload.

Many authors have designed real-time scheduling algorithms that are resilient to the effects of system overload [6, 8, 19]. For example, Buttazo, et al. [5] present a comparative study of scheduling algorithms, which characterize tasks not only according to their deadline but also according to an importance value. They show, notably, that the best results are obtained when scheduling is based on deadline before overload and on importance value during overload.

However, designing algorithms to manage overload in RTDBSs has received comparatively little attention and the few efforts in this area considered only centralized architectures. Pang, et al. [24] designed an algorithm that exhibits the good performance of EDF (Earliest Deadline First), with the assumption that the average completion ratios of the different transaction classes, characterized by their mean sizes, being the same. Datta, et al. [9] developed an admission control mechanism, which handle transient overloads while guaranteeing access to some particular classes of transactions. As in [24], classes are characterized by their mean sizes and the goal is to ensure fairness between transaction classes. Bestavros, et al. [4] consider overload management for soft deadline transactions where primary transactions have compensating transactions. An admitted transaction is guaranteed to complete its execution either by successful commitment of its primary transaction or by safe termination of its compensating one. In [13], Hansson, et al. have presented the value-driven Overload Resolution (OR) algorithm for handling dynamic multi-class workloads. The algorithm handles critical sporadic transactions having alternative/contingency transactions and non critical aperiodic firm-deadline transactions by reallocating resources. However, the performance evaluation of the algorithm showed that, during transient overloads, the latter degrades the performance of non-critical transactions, when the timeliness of critical transactions is enforced. To this end, Hansson et al. introduce in [14] a variation of OR, denoted OR-ULD (Overload Resolution – Utility Loss Density). They add a novel value-driven bias control mechanism which enables the algorithm to enforce robustness constraints by performing bias control between transaction classes. In [15], they evaluate OR-ULD algorithm for imprecise sporadic-task workloads, where tasks are decomposed into one mandatory task and one optional task. In another context, Carney, et al. in [7] have developed at Brandeis University, Brown University, and M.I.T, a Data Base Management System (DBMS) called Aurora, which handles applications that acquire data from external sources (e.g., sensors) rather than from humans issuing transactions. The role of the DBMS in this context is to alert humans when abnormal activity is detected. Aurora integrates a QoS evaluator that continually monitors system performance and activates a load shedder upon overloads. The load shedder remains active until the performance of the system reaches an acceptable level. Recently, Amirijoo, et al. [1] have proposed a platform that specifies and manages the quality of service (QoS) in imprecise real-time databases. In their approach, they use not only the notion of data imprecision but also that of transaction imprecision in order to obtain a more efficient management of QoS and overload. Their concept data imprecision is similar to ours. On the other hand, the concept transaction imprecision assumes that any transaction includes a mandatory part which must be executed within the deadline, and an optional part which is executed only if required resources are available (such as processor, time). The authors

---

1 The admission control determines which transactions should be granted system resources.
define QoS according to the rate of mandatory/optional transactions that miss their deadline, to the rate of discarded transactions, and to other parameters. Update transactions are discarded when the new value to write is very close (is in the interval of authorized deviation) to the value stored in the database. In our case, this write transaction is not discarded and read transactions are allowed to execute in parallel with it.

References


Control for Real-Time Replicated Database Systems,” 5th Int. Conf. on Enterprise Information Systems, Angers, France, April 2003.


Samia Bouzefrane (http://cedric.cnam.fr/~bouzefra) is an associate professor at the Conservatoire National des Arts et Métiers (CNAM, Paris). She received her Ph. D. in Computer Science in 1998 from the University of Poitiers (France). She joined the CEDRIC Laboratory of CNAM on September 2002 after 4 years at the University of Le Havre. Currently she works on different research domains such as real-time database systems, the use of the component approach in the design of real-time and/or embedded applications; and more recently on a project called MESURE (http://mesure.gforge.inria.fr/) to benchmark Java Card platforms. She received on September 2007 the Isabelle Attali Award from INRIA, which honours the most innovative work presented during “e-Smart” Conference. Furthermore, she is the author of two books : the first one is a Computer-Science dictionary in French, English and Berber (Ed. L’harmattan, 1996) and the second one deals with the operating systems (Systèmes d’exploitation: cours et exercices corrigés Unix, Linux et Windows XP avec C et Java, Ed. Dunod, 2003, 566 pages).

Jean-Paul Etienne has obtained his Master degree in Safety Engineering and Formal Methods at Paris 6 /CNAM in 2003. He is currently completing his PhD at the CEDRIC laboratory CNAM. The presented work is part of the result obtained during his internship at the laboratory in fulfillment of his Master degree. His current research activity deals mainly with applying the Component Based Software Engineering paradigm in real-time and embedded systems.

Claude Kaiser (http://cedric.cnam.fr/~claude/) is Professor Emeritus of CNAM at the CEDRIC Laboratory. He has worked for long on real-time systems (system design for the French Marine, scientific consultant for Cerci/Peugeot and for the system Dune_iX), operating systems (work done at INRIA), and distributed systems (scientific consultant for Chorus OS). Currently, his main research activity pertains to verification and validation of concurrent applications. A tool called Quasar, based on Petri-nets has been developed for this purpose. He is also the co-author of several books on programming paradigms and algorithms (Grégoire), operating systems (Crocus, Cornafion) and real-time systems (Real-time Scheduling, Hermès 2000 et Scheduling in Real-Time Systems, Wiley, 2002).