A Java Platform to Control Real-Time Transactions Overload

Jean-Paul Etienne and Samia Saad-Bouzefrane
Laboratoire CEDRIC, Conservatoire National des Arts et Métiers
292, rue Saint Martin, 75141, Paris, FRANCE
samia.bouzefrane@cnam.fr

Abstract

Current applications are distributed in nature and manipulate time-critical databases with firm-deadline transactions. A transaction submitted to a master site is splitted into subtransactions executed on participant sites which manage each a local database. In this paper, we propose a Java platform based on a protocol that manages real-time distributed transactions with firm-deadline, in the context of possible overload situations and imprecise data acceptable utilization.

1. Introduction

Current applications, such as Web-based services, electronic commerce, mobile guidance by telecommunication systems, multimedia applications, etc. are distributed in nature and manipulate time-critical databases. In order to enhance the performance and the availability of such applications, the major issue is to develop protocols that manage efficiently real-time transactions while tolerating overload in the distributed system. In fact, if the system is not designed to handle overloads, the effects can be catastrophic and some primordial transactions of the application can miss their deadlines. While many efforts have been made in the management of transient overloads in centralized Real-Time Database Systems (RTDBSs) [5, 2, 6] few works control the overload in a distributed environment.
Moreover, the objective of maintaining logical consistency in the database is difficult to reach ; some proposed works attempt to relax the isolation transaction property, i.e., serializability, usually considered as the transaction correctness criteria in traditional database management systems [4]. Indeed, there exists many kinds of applications where strict serializability is not necessary; hence, that may tolerate data imprecision. For example, in applications where approximate results obtained may be more useful than accurate ones obtained late. In this paper, we focus on the design of a Java platform based on a commit processing protocol that cares of overload situations and that bounds database logical inconsistencies. The overload occurs when the computation time of transactions set exceeds the time available on the site processor and then the deadlines can be missed. ε-data concept initially proposed in [11] is employed here as a correctness criterion to guarantee the consistency of the distributed database. Our study is concerned by "firm-deadline" transactions because many current applications such as Web-based services use communication protocols with timeout features. In firm-deadline applications, each transaction that misses its deadline is useless and then aborted immediately. The remainder of this paper is organized as follows : Section 2 presents the related work. Section 3 describes the database model used. Section 4 introduces a notion close to epsilon serializability called ε-data that allows more execution concurrency between transactions. In Section 5 is described the mechanism used to control transactions overload. Section 6 examines the principle of the protocol implemented by our simulation platform. Section 7 presents the Java platform. Experimental results show good performances under overload and ε-data conditions in Section 8. Section 9 concludes the paper.

2. Related work

Many authors have designed real-time scheduling algorithms that are resistant to the effects of system overload [3, 7]. However, designing algorithms to manage overload in RTDBSs has received comparatively little attention and the few efforts in this area have assumed a centralized real-time database system. For example, Hansson et *al.* in [5, 6] propose an algorithm denoted OR-ULD (Overload Resolution-Utility Loss Density) that resolves transient overloads by rejecting non critical transactions and by replacing critical ones with contingency transactions. Bestavros et *al.*, in [2], consider overload management for soft-deadline transactions where primary transactions have compensating transactions. Transactions are guaranteed to complete either by successful commitment of the primary transaction or by safe transaction of the compensating transaction. Among the techniques that use the concept of importance, Saad et *al.* have proposed a protocol to control the transactions load in a replicated RTDBS [13] and another overload-management protocol in a non replicated RTDBS where transactions tolerate data imprecision [12]. Transactions are assigned values used to define the importance degree of each transaction with respect to the application. In order to decrease the transactions load, only the transactions declared "important" by the application developer have their execution maintained, the other transactions considered as less important are rejected. In [12], we have applied this principle to resolve transient overloads that may occur in distributed real-time database systems. Furthermore, to increase concurrency execution between transactions without loss of data consistency we relax ACID properties (Atomicity, Consistency, Isolation, Durability) judged too restrictive in real-time context [10]. Researchers have proposed techniques that take into account data semantics to relax these properties. These works have led to the development of forced wait and data similarity policies [15] and epsilon serializability criterion [9]. In the next section, we present the database model used and then we recall the principle of ε-data concept used to relax data properties in [11]. Besides the overload control, the protocol proposed in [12] integrates the ε-data concept in order to allow more transactions concurrency and therefore to reduce the number of firm transactions that miss their deadlines. In this paper, we recall the principle of the protocol proposed in [12] to control overload in a distributed database where transactions may manipulate imprecise data. Then, we describe the Java simulation platform, based on this protocol, that we have developed.

3. The database model

In our model, we consider only firm real-time transactions. A transaction is submitted to the master site where is executed the *coordinator* process. The result of the transaction is delivered before transaction deadline. Distinct parts of the database are located on different sites. The global transaction is decomposed into subtransactions that are sent to the participant sites, where they are managed by a *cohort* process. Each participant site includes three modules (see Figure 1):

- a scheduler module that uses EDF scheduling algorithm [8],
- an overload-manager module that controls transient overloads each time a new subtransaction is inserted in the ready-transactions queue and
- a data-manager module that applies ε-data concept when subtransactions of the participant site conflict while accessing data.

The model does not consider database replica. Two types of subtransactions are allowed in our environment: query subtransactions and update subtransactions.

4. ε-data notion

ε-data, is a notion close to epsilon serializability [9]. It introduces some levels of data-imprecision in order to deal with real-time applications that tradeoff consistency for timeliness, provided the amount of inconsistency introduced can be controlled. In multi-media applications, the loss of some packets (composing an audio or a video stream) may not lead to serious consequences. The ε-data concept deals with data absolute-value imprecision which is a percentage of the current value of the data. For example, let a data item d = 20. If d tolerates an imprecision ε then it is called ε-data. Let ε = 10%, then we tolerate the use of a value which is in the interval [20-2, 20+2]. We denote by ε-data, a data item whose exact value is v and where each value in [v-ε , v+ε] is acceptable. In strict context, if a write transaction already holds a lock on a data item d, a query transaction requesting a lock on d will wait or will be aborted. In ε-data applications, the isolation level tolerated is not maximal, that is, each read transaction may access a data while another transaction is writing it provided that the quantity of inconsistency introduced by the write transaction is less than a specified bound, ε. Therefore, locks management used by transactions is somewhat different from the one used in classical RTDBSs.

5. Overload Management

The arrival of a new subtransaction may cause an overload, and thus a timing fault, if the required computation times and deadlines exceed the computing capability of each site processor to fulfil all subtransactions deadlines. Our overload management policy aims to favour the execution of the most important transactions of the application. A favoured transaction must undergo less timing faults (i.e., deadline misses) and less abortions due to overload than other transactions. It is then necessary to have a means to designate the most important transactions. This means is presented in the form of a parameter called importance value as described in [13].

5.1. Transaction Importance

Each global transaction is associated with a positive integer that represents the importance value of the transaction in the transactions set. The importance value is intrinsic to the application, so it can be fixed by the application developer. For example, in the field of electronic stock exchange, the transaction that has to update quotations in real-time is more important than the one that simply read quotations. Each transaction is characterized by a deadline which defines its urgency and by an importance value which defines the criticality of its execution, with respect to the other transactions of the real-time application.

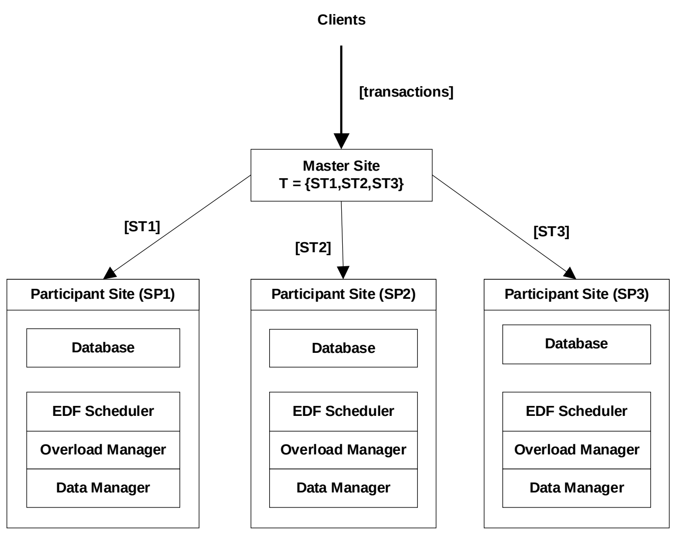

Figure 1. The database model

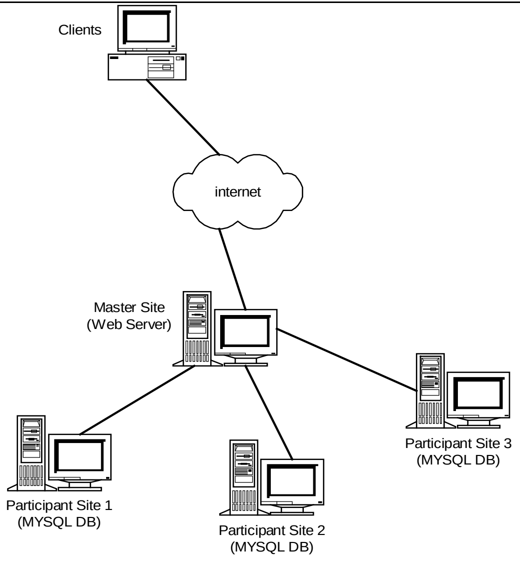

Figure 2. The simulation platform architecture

The importance of a transaction is not related to its deadline; thus two different transactions which have the same deadline may have different importance values. In our model, a global transaction is characterized by its arrival time, its deadline and its importance value. In the same way, within a site a subtransaction is characterized by its arrival time, its execution duration, its deadline and its importance value. Note that the deadline and the importance value of a subtransaction are inherited from the global transaction to which it belongs.

5.2. Stabilization process

The stabilization process aims to manage system overload and consists in privileging the transactions that have high importance values. Transactions that have the lowest importance values are released from the ready-transactions queue and are 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.

6. Protocol description

Our simulation platform is based on a protocol designed to manage distributed real-time transactions. It focuses on firm real-time transactions and uses the model described in Section 3. This protocol, proposed in [12], manages overload within each participant site while transactions manipulate ε-data. It is based on the 2PC (two Phase Commit) [14, 1] protocol for the communications between sites.

6.1. Integrating overload control

The protocol used by our simulation platform augments 2PC protocol in order to handle overload conditions. When the coordinator process receives a transaction Ti to execute, it splits it into subtransactions. For each subtransaction STij, the coordinator sends a message *INITIATE(STij)* to execute STij on the cohort process that manages data items needed by STij. When the cohort receives an *INITIATE(STij)* message, it applies the stabilization process. That is, if the site is overloaded because all the local subtransactions cannot be executed on time, the ready-transactions queue is stabilized by aborting the subtransactions that have the lowest importance values. When a subtransaction is aborted, the cohort sends a "NO" vote to the coordinator. As soon as it receives a "NO" vote, the coordinator broadcasts ABORT messages to all the cohorts for invalidating the local subtransactions. On the other hand, if the coordinator receives YES messages from all its cohorts then it broadcasts to them COMMIT messages.

6.2. The stabilization process description

The conditional laxity $LC_{STi}(t)$ of a transaction *STi* of the ready queue is the maximum time during which *STi* may be delayed without missing its deadline, with the assumption that all transactions with earlier deadline will finish their execution before *STi* may start running. The processor laxity LP is defined as a minimal value of the conditional laxity of each transaction of the ready queue. An overload situation is detected as soon as the site laxity LP(t) is less than 0. The late transactions are those whose conditional laxity is negative. The overload value is equal to the absolute value of the processor laxity, |LP(t)|. The stabilization process consists in privileging, when the site is overloaded, transactions with high importance values. For this purpose, we remove from the $readyQueue_{s,t}$ the transactions that have the lowest importance values until the laxity is positive anew. Moreover, we should not remove a transaction *ST* from $readyQueue_{s,t}$ when this queue contains transactions that have lower importance values than *ST* and which rejection would have resulted in a positive laxity.

6.3. Integrating ε-data concept in the locking condition

Each time a subtransaction is submitted to a cohort, the site-processor laxity is computed. If the laxity is negative, one or more subtransactions with low importance values are aborted in order to maintain a positive laxity. Moreover, to increase concurrency between the subtransactions within a participant site, we apply the ε-data concept. The locking condition concerns a situation where a data item *d* is write-locked by a W^{ε} transaction and a query transaction Q_i^{ε} requests to read-lock *d*. Instead of blocking or aborting Q_i^{ε} transactions as in classical protocols, in our protocol all Q_i^{ε} transactions pursue their execution concurrently with W^{ε} provided that the difference between the value written by W^{ε} and the value read by Q_i^{ε} transactions does not exceed ε. Otherwise, if the value written by W^{ε} is out of range then the transaction manager behaves classically. To increase concurrency between transactions, ε-data concept operates at two levels :
- *at the execution phase of a write transaction* : all read transactions that request for a data item write-locked by a transaction that is in its execution phase, may execute in parallel with the write transaction provided that the data-item inconsistency is bounded by ε.
- *at the uncertainty phase of a write transaction*: the uncertainty phase of a transaction begins at the time when it finishes its execution and remains waiting for a COMMIT or an ABORT message. Due to message exchanges, this uncertainty phase may last some significant time. Hence, if a write-locked data item provides a bounded inconsistency then all the read transactions that request this data item will access to its value and continue their execution in parallel with the write transaction. During the uncertainty phase, read locks are released while write locks are kept until the transaction validation.

7. The Java simulation platform

The simulation platform is based on Java technology and makes use of MySQL databases. Its architecture is composed of a master site and three participant sites over which the database system is distributed. Transactions are sent via HTTP requests to the master site which splits them into subtransactions and distributes them to appropriate participant sites, where they are processed into SQL statements and executed. In addition, transactions are composed only of read and update operations to ensure that the ε-data concept is applied during database access. Furthermore, the communication framework is based on socket primitives, rather than CORBA, for performance reasons and messages are modelled as objects. An overview of the architecture is shown Figure 2.

7.1. Master Site

The master site is implemented as a Java Servlet and sits on a TomCat Server. Transaction requests are made via HTTP and each request is handled by an instance of the Servlet. A Transaction request contains one or more data operations and each operation includes: 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 about the participant site which handles the record. As shown in Figure 3, the master site distributes subtransactions to participant sites via *INITIATE* messages. The attributes of these messages are determined as follows:
- the number of the global transaction is calculated as an addition of the arrival time of the transaction at the master site and a random number ranging from 0 to one million.
- the number of each subtransaction is represented by either 0 or 1 or 2. Each number represents a participant site to which the subtransaction will be sent (0 for the participant site 1, 1 for participant site 2 and 2 for participant site 3).
- the transaction deadline is calculated from the arrival time of the transaction at the master site and the execution time of all its data operations. The execution time of each data operation is determined by its type (read or write). An additional time is also included to cater for communication time between sites.

7.2. Participant site

Each participant site uses the following queues :
- *ReadyQueue* contains a list of ready transactions sorted in ascending order, based on their deadline. Thus, the first element in the queue represents the transaction which has the nearest deadline.

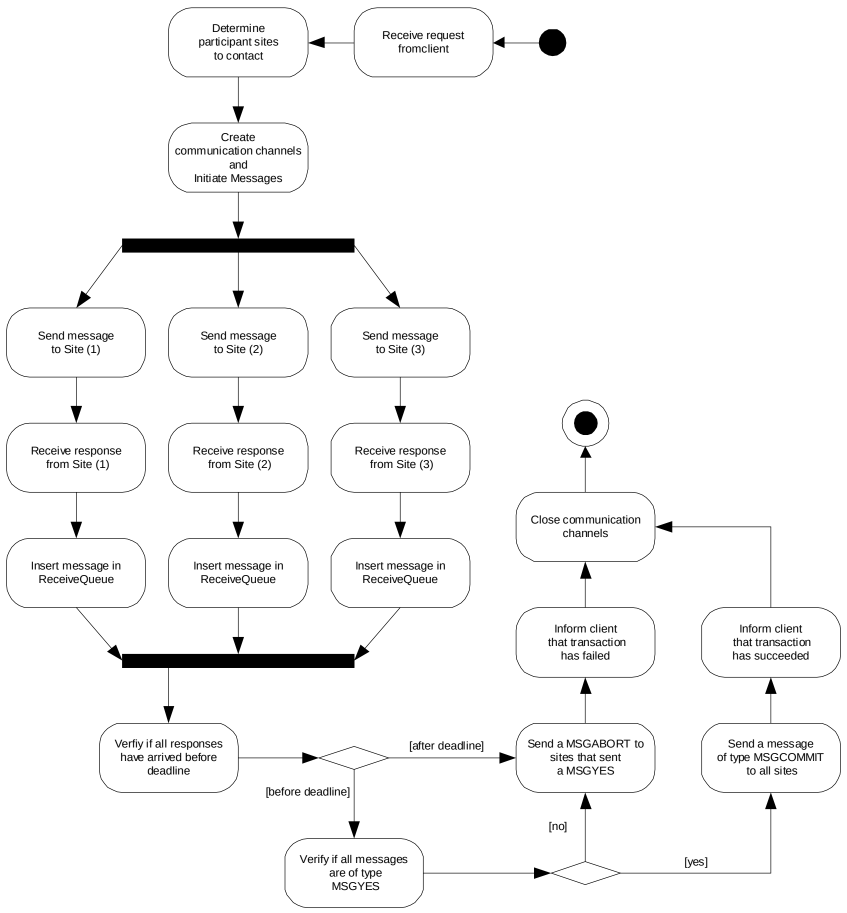
Determine participant sites to contact
Receive request fromclient
Create communication channels and Initiate Messages
Send message to Site (1)
Send message to Site (2)
Send message to Site (3)
Receive response from Site (1)
Receive response from Site (2)
Receive response from Site (3)
Insert message in ReceiveQueue
Insert message in ReceiveQueue
Insert message in ReceiveQueue
Verfiy if all responses have arrived before deadline
[after deadline]
[before deadline]
Verify if all messages are of type MSGYES
[no]
[yes]
Send a MSGABORT to sites that sent a MSGYES
Send a message of type MSGCOMMIT to all sites
Inform client that transaction has failed
Inform client that transaction has succeeded
Close communication channels

Figure 3. The activity diagram of the master site

- *ImportanceQueue* contains a list of ready transactions sorted in ascending order, based on their importance value. The first element in the queue represents the least important transaction.
- *UncertainQueue* contains transaction elements which have completed their execution and which are waiting for the commit/abort decision of the master site.
- *ReceiveQueue* contains newly arrived transactions at the participant site.
- *WaitQueue* contains transactions that have been suspended due to conflicting data access.

The participant site is composed of various modules, implementing the Overload Manager, the EDF Scheduler and the Data Manager. All these modules are defined like threads and work concurrently on the different queues (see Figure 4). They also share a connection to the local database. Besides, accesses to these entities are mutually exclusive to ensure data consistency. The *OverloadManager* module applies the stabilization process at the arrival of new subtransactions stored in the *ReceivedQueue*, to detect and resolve overload situations. Every request received by the participant site via an *INITIATE* message is handled by an instance of *ServiceThread* module which transforms it into a transaction object and inserts it into the *ReceivedQueue*. The *Scheduler* and the *Worker* modules implement the EDF Scheduler. Moreover, the *Worker* acts as the data manager, during the execution of a subtransaction, by applying the ε-data concept to determine the execution pattern to adopt in the face of locked data items. The *ResponseManager* and the *ResponseThread* modules work together to inform the master site that subtransactions have finished their execution and wait for its decision to determine whether to commit or to abort them.

7.3. Locks handling

To handle locked data items, we have made use of two tables, *finaldata* and *tmpdata*. Permanent records are stored in *finaldata*, while temporary records are stored in *tmpdata*. Since accesses to the database are mutually exclusive, locked data items are determined during the application of the ε-data concept by querying the table *tmpdata*. This table contains records representing write-locked data items held by subtransactions during their execution.

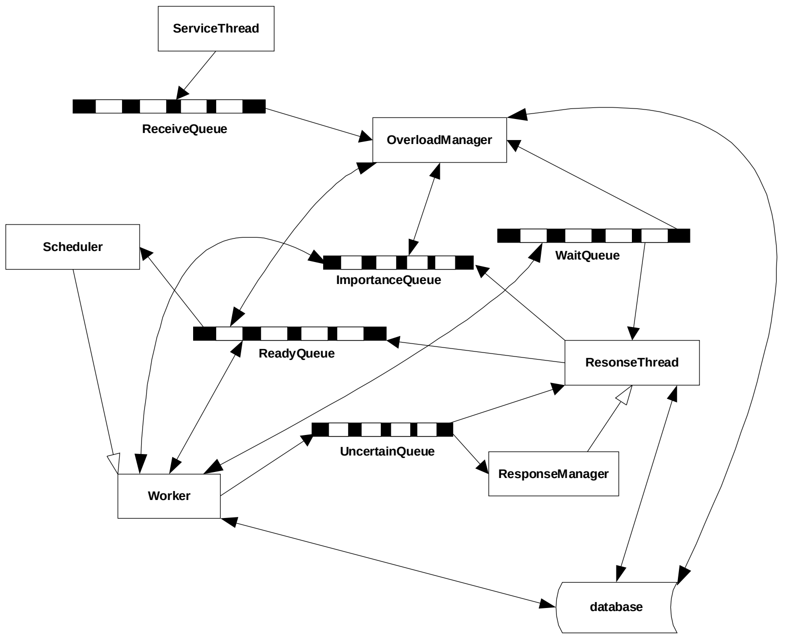

Figure 4. The architecture of a participant site

Besides, these records remain in the table as long as the subtransactions that hold them are not committed or aborted yet. When a subtransaction is committed, records that represent its write-locked data in the *tmpdata* table are used to update the *finaldata* table before being deleted. Furthermore, when a data record is not write-locked, a read operation on this record is performed on *finaldata* table, otherwise it is executed on *tmpdata* table provided that the after-value of the data item lies in an acceptable ε-data range.

8. Experimental results

The application developed to evaluate the performance of our simulation platform under various working conditions is composed of three modules:
- *Execution* Module,
- *Configuration* Module and
- *Statistics* Module.

8.1. The Execution module

Once started, the *Execution* module gets the transactions to be executed from a file and sends them simultaneously, in the form of HTTP requests, to the master site. It then waits for the execution results sent through the replies of the master site and stores them in a database for later analysis. Each result contains the following information:
- the transaction number,
- the execution time of the transaction,
- the importance value of the transaction and
- the execution result (commit/abort).

A name is associated to each series of execution during their recording in the database in order to distinguish them during analysis. Consequently, during analysis, this name enables the statistics module to group the series of execution according to the type of configuration under which they have been executed.

8.2. The Configuration module

This module enables the user to change the execution parameters of the distributed system. These parameters are presented as follows:
- *ReadTime* : defines the execution time of a read operation,
- *WriteTime* : defines the execution time of a write operation,
- *AddTime* : defines the additional time that is added to the transaction execution time to determine its deadline,
- *ImpRead* : defines the importance value of a transaction which is composed of read operations only,
- *ImpWrite* : defines the importance value of a transaction, which is composed of write operations only,
- *ImpReadWrite* : defines the importance value of a transaction which is composed of both read and write operations,
- *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 the stabilization process.

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

8.3. The Statistics module

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

- Percentage of transactions meeting their deadline, grouped by series of execution.
- Percentage of transactions meeting their deadline, grouped by their importance value.
- Efficiency of transactions, which is represented by $(\sum C_{Ti})/d$ where C_{Ti} represents the execution time of each transaction T_i and d represents the duration of the experiment.

8.4. Simulation analysis

During the simulation phase, 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, 300) in order to allow one to evaluate the behavior of each simulation configuration vis-à-vis a linear increase of the number of requests. The series have also 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 10% 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 participant sites, each having a local database of 30 records. Also, each distributed transaction contains three sub-transactions, one for each participant site, and each sub-transaction has at most 3 data operations. In addition, the database records are set back to their initial value at the end of each series of execution in order to ensure that each test does not take advantage of the modifications brought by previous simulations. Since transactions are sent concurrently to the master-site, their order of arrival and execution may differ from one series of execution to another. As, this may affect the results obtained, the series are executed several times for each configuration and the final result is computed as the mean of the results of these executions.

Simulation tests were carried out, at first, to evaluate separately the performance of the ε-data concept and that of the importance value. Thereafter, several tests have been performed to appraise the system performance under various conditions.

8.4.1. **ε-data concept test.** This test has been realized using the ε-values 0%, 5%, 10%. The *Addtime* parameter has been set to a relatively high value (90000) in order to prevent occurrences of overload situations. Hence, this ensured that transactions could be aborted only during data access conflicts, which allowed us to have a better appraisal of the performance of the system for each ε-value used. The results presented in Figure 5 show that an increase of the ε-value entails a net increase of the percentage of transactions meeting their deadline. This shows clearly that the use of ε-data concept augments considerably concurrent executions of transactions.

8.4.2. **Tests combining ε-data values and importance values.** The tests presented in Figure 6 have been carried out to evaluate the system under various configurations, comprising the ε-data concept and that of the importance value. The tests have been performed under four configurations:

- when the importance value of transactions is considered and the ε-data concept is applied,
- when the importance value of transactions is considered and the ε-data concept is not applied,
- when the importance value of transactions is not considered and the ε-data concept is applied and finally
- when the importance value of transactions is not considered and the ε-data concept is not applied.

Furthermore, these tests have been carried out in a working environment where occurrences of overload situations are frequent (*Addtime*=3500) and where the ε-value = 10% when the ε-data concept is applied.

The results of Figure 6 show that the system achieves the best performance when the ε-data concept is applied and when the importance value of transactions is considered during the stabilization process.

9. Conclusion

In this paper, we have focused on the design of a simulation platform based on a commit processing protocol that bounds database logical inconsistencies and that manages transient-overload situations of the distributed system. When an overload is detected within a participant site, the transactions that are important for the application are favoured. The less important ones are discarded from the system. ε-data concept is employed here as a correctness criterion to guarantee the consistency of the distributed database. The simulation platform is based on Java technology and makes use of MySQL databases. Transactions are sent via HTTP requests to the master which is implemented as a Java Servlet on TomCat server. Each participant site implements an overload manager, an EDF scheduler and a data manager. This platform integrates a graphical interface to submit transactions, a configuration module to fix a certain number of parameters and a statistical module that displays the simulation tests in a graphical way. The experimental results show good performances under overload and ε-data conditions. As a perspective to our work, we intend to use a model with dynamic importance-values and to integrate to our protocol an optimistic concurrency control.

10. References

[1] P. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems", Addison Wesley, 1987.

[2] A. Bestavros and S. Nagy,"Value-cognizant Admission Control for RTDB systems", *Proc. of the 17th Real-Time Systems Symp.*, pp. 230-239, IEEE Computer Society, dec. 1996.

[3] G. C. Buttazo, G. Lipari and L. Abeni, "Elastic Task Model for Adaptive Rate Control", in *Proc. of IEEE Real-Time Systems Symposium*, Madrid, dec. 1998.

[4] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The notion of consistency and predicate locks in a database system", CACM, **9(11)**, pp. 624-633, 1976.

[5] J. Hansson, S. H. Son, J.A. Stankovic and S. F. Andler,"Dynamic Transaction Scheduling and Reallocation in Overloaded Real-Time Database Systems", *Proc. of the 5th Conference on Real-Time Computing Systems and Applications (RTCSA'98)*, pp. 293-302, IEEE Computer Press, 1998.

[6] J. Hansson and S. H. Son,"Real-Time Database Systems: Architecture and Techniques", *K. Lam and T. Kuo (eds.)*, Kluwer Academic Publishers, pp. 125-140, 2001.

[7] G. Koren and D. Shasha, "Dover: An Optimal On-Line Scheduling Algorithm for Overloaded Uniprocessor Real-Time Systems", *SISAM J. Comput.*, **24(2)**, pp.318-339, 1995.

[8] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the ACM, Vol. 20, n°1, pp. 46-61, 1973.

[9] C. Pu, "Generalized Transaction Processing with Epsilon Serializability", in Proc. of the 4th Int. Workshop on High Performance Transaction Processing, sept. 1991.

[10] Ramamritham K., "Real-Time Databases", J. of Distributed and Parallel Databases, Vol. 1, n° 2, pp. 199, 226, 1993.

[11] S. Saad-Bouzefrane and B. Sadeg, "Relaxing the Correctness Criteria in Real-Time DBMS", Int. Journal of Computers and their Applications, 7(4), pp. 209-217, 2000.

[12] S. Saad-Bouzefrane, "How to Manage Real-Time Transactions Overload in ε-data Applications ?", EuroMicro RTS Conference, WIP session, June 2003, Portugal.

[13] S. Saad-Bouzefrane and C. Kaiser, "Distributed Overload Control for Real-Time Replicated Database Systems", 5th Int. Conf. on Enterprise Information Systems, April 2003, Angers, France.

[14] G. Samaras et al., "Two-Phase Commit Optimization in a Commercial Distributed Environment", Journal of Distributed and Parallel Databases, 3(4), 1995.

[15] M. Xiong, J. A. Stankovic, K. Ramamritham, D. Towsley and R. M. Sivasankara, "Maintaining Temporal Consistency : issues and algorithms", 1st Int. Workshop on RTDBS: Issues and Applications, pp. 1-6, California, 1996.

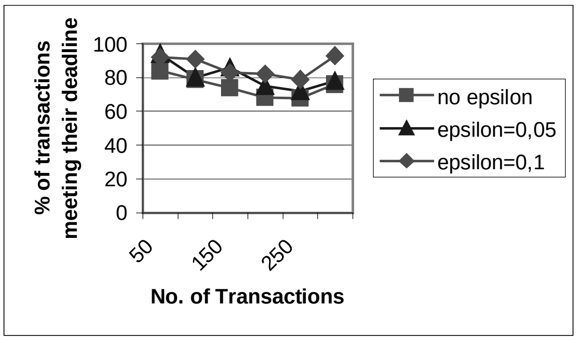

Figure 5. The percentage of transactions meeting their deadline when considering ε-data concept

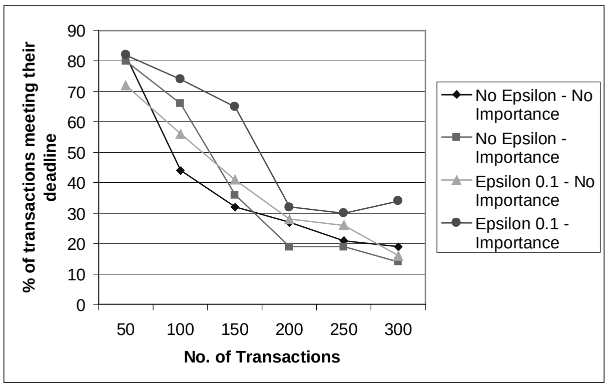

Figure 6. Percentage of transactions meeting their deadline in four situations