How to Manage Replicated Real-Time Databases in an Overloaded Distributed System ?

S. Saad-Bouzefrane - C. Kaiser
Laboratoire CEDRIC, Conservatoire National des Arts et Métiers,
292 rue Saint Martin, 75141 Paris Cédex 03, FRANCE
Tel: +33 (1) 40 27 25 83 - Fax: +33 (1) 40 27 22 96
{samia.bouzefrane, kaiser}@cnam.fr

Abstract

In order to meet their temporal constraints, current distributed applications such as Web-based services and electronic commerce use the technique of data replication. To take the replication benefit, we need to develop concurrency control mechanisms with high performance even when the distributed system is overloaded. In this paper, we present a protocol that uses a new notion called {\it importance value} which is associated with each real-time transaction. Under conditions of overload, this value is used to select the most important transactions with respect to the application transactions in order to pursue their execution ; the other transactions are aborted. Our protocol RCCOS (*Replica Concurrency-Control for Overloaded Systems*) augments the protocol MIRROR, a concurrency control protocol designed for firm-deadline applications operating on replicated real-time databases in order to manage efficiently transactions when the distributed system is overloaded. A platform is currently being developed to measure the number of transactions that meet their deadlines when the processor load is controlled.

Key-words: Firm real-time transactions, distributed database system, commit processing, processor overload, data replication.

1. Introduction

Current applications, such as Web-based services, electronic commerce, mobile telecommunication system, etc., are distributed in nature and manipulate time-critical databases. In order to enhance the performance and the availability of such applications, one of the main techniques is to replicate data on multiple sites of the network. Therefore, the major issue is to develop efficient replica concurrency control protocols that are able to tolerate the overload of 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 transactions for replicated databases in the real-time context [21, 22, 23, 24, 25], no work deals with protocols that manage distributed real-time databases and simultaneously control the overload of the system. Some researches have dealt with the scheduling of tasks under overload conditions when the real-time system is centralized such as in [9] and [16], or distributed as in [15]. In [10], a protocol that controls the processor overload has been integrated in RT-Linux.

In this paper, we focus on the design of {\it one-copy serializable} concurrency control protocols for replicated real-time databases when the distributed system is overloaded. The overload occurs when the computation time of transactions set exceeds the time available on the processor and then the deadlines can be missed. 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 is then aborted immediately.

We present in this paper, a replica concurrency-control protocol that manages the overload on each site of the distributed system by choosing only the "most important" transactions with respect to the application transactions, to maintain their execution ; the other transactions are simply aborted. This principle can be applied, for example, to Web applications that provide free and payable services. The client requests correspond to transactions executed on a distributed database management system.

If the system is overloaded, the client requests that concern payable services are marked as more important than those dealing with free services.

In the protocol we propose here, an *importance value* is defined for each transaction in order to distinguish the most important transactions for which it is essential to maintain the execution from the rest of transactions that may be aborted. Our aim is to alleviate the processor load. The importance criterium is used only to control the processor load since that ready transactions of each site are scheduled using a local scheduling algorithm such as EDF algorithm [18]. Our protocol RCCOS (*Replica Concurrency-Control for Overloaded Systems*) augments MIRROR (*Managing Isolation in Replicated Real-time Object Repositories*) [25] a concurrency control protocol designed for replicated real-time databases that adds data conflict-resolution mechanisms to the optimistic two-phase locking (O2PL) [7] designed for a non real-time context. An experimental platform is being developed to implement RCCOS and MIRROR protocols in order to compare their performance under conditions of overload.

The remainder of this paper is organized as follows: Section 2 presents the related work. In Section 3, is described the model used whereas in Section 4 are described RCCOS protocol and MIRROR protocol on which it is based. The protocol implementation is described in Section 5. Some elements of proof are given in Section 6. Before concluding the paper in Section 8, we present an experimental platform, in Section 7, that is being developed to measure the performance of our protocol.

2. Related work

The protocols designed to manage distributed transactions in replicated databases [2, 3, 17, 20] are not suitable to real-time context because they are prone either to blocking or to message overhead or to both. This is incompatible with the respect of real-time constraints. Authors (eg., [21, 22, 23, 24]) have then begun to work on concurrency control for replicated Distributed Real-Time Data Base Systems (DRTDBS) and have designed protocols that take into account the transactions real-time constraints. For example, a multi-version technique is proposed in [21] to increase the degree of concurrency. In [22, 23], a real-time scheduling is integrated in replication control. While no work has dealt with the management of distributed real-time transactions in overloaded systems, some authors have designed algorithms that resolve overload in real-time database systems [11, 12] and others have studied the resistance of real-time scheduling algorithms to the effects of system overload [8]. In fact, some algorithms deal with periodic task sets and allow the system to handle variable computations times which cannot always be bounded [1, 19, 6]. Other algorithms deal with hybrid task sets where tasks are characterized with an importance value [9, 16, 15]. All these policies deal with overloads to provide deadline missing tolerance. In [4], Baruah et al. propose ROBUST an on-line uniprocessor scheduling algorithm that performs for a large range of slack factors during overload. Slack factor is defined as the ratio between the task deadline and its execution time.

Among the techniques that use the concept of importance, Kaiser et al., in [15], have proposed a protocol to control the tasks load in a distributed real-time system. Tasks are assigned values used to define the importance degree of each task with respect to the application-tasks set. In order to decrease the tasks load, only the tasks declared "important" by the protocol have their execution maintained, the other tasks considered as less important are aborted.

Delacroix et al., in [10], have implemented an overload controller that has been integrated to RT-Linux. In their task model proposed, each task is made up of several execution modes. The *normal mode* is a mode executed when the task begins to execute. It takes care of the normal

execution of the task. And the *survival modes* are executed when the task is cancelled by the overload resorption or when it misses its deadline.

3. The model

In our model, we consider only firm real-time transactions, that is, only transactions that fully complete execution before their deadlines are considered to be successful. On the contrary, transactions that miss their deadlines are considered worthless and are immediately discarded without being executed to completion [13].

The distributed system is composed of :

- a master site on which each global transaction is submitted and
- sites that have replicas of parts of a distributed database.

Each submitted transaction T_i is decomposed into n subtransactions ST_{ij} that are sent to sites called *participant sites*. All these subtransactions are mandatory, that is, T_i commits only if all its subtransactions commit locally within their sites. On each participant site, subtransactions are managed by a *cohort process*. The participant site of a subtransaction ST_{ij} is the one that manipulates the least important transactions among the sites that have replicas of data-items needed by ST_{ij}. The sites that are not participant sites but that have replicas of the database are called *updaters* (see Figure 1). All the sites, including the master, are partially connected, but the graph representing the network is a connected graph, that is, from any site it is possible to access another site directly or indirectly. A global transaction is characterized by its arrival time and its deadline while, within a participant site, a subtransaction is characterized by its arrival time, its execution duration and its deadline. The execution time must be lower than the relative deadline. Moreover, the subtransaction deadline is identical to the deadline of the global transaction to which the subtransaction belongs.

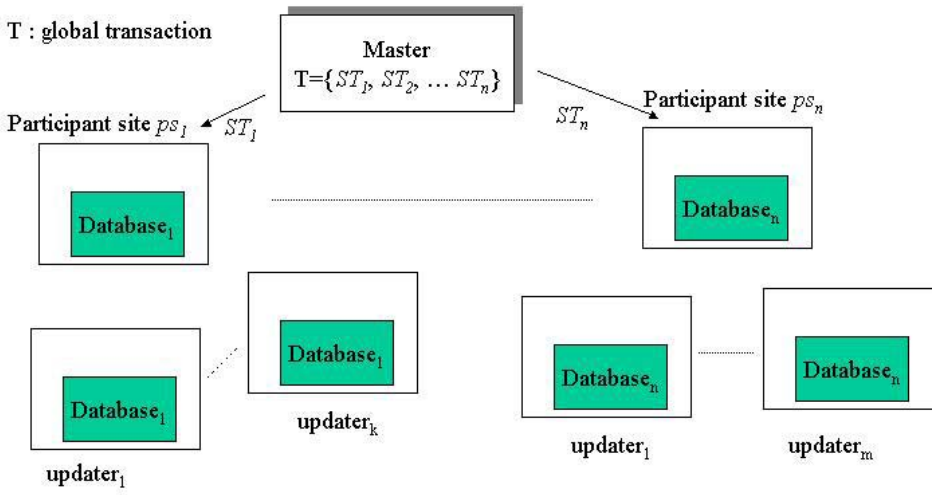

Figure 1: *A replicated distributed database system*

4. Protocol description

Since our protocol RCCOS augments the MIRROR protocol in order to handle the distributed-system overload, we first review the principle of MIRROR before describing the proposed protocol.

4.1 MIRROR protocol

The MIRROR protocol described in [25] extends the implementation strategy for centralized optimistic concurrency-control algorithms proposed in [14] to handle data distribution and replication.

Transactions follow three steps during their execution: read, validation and write. In the read phase, cohorts acquire locks to access data items on their local sites. The updating of replicas is deferred to the end of transaction, that is, to the commit process.

In the validation phase, a cohort that receives a PREPARE message from its master begins a local validation. If it fails, it sends an ABORT message to the master. Otherwise it sends PREPARE messages containing relevant updates to its updater sites. Each updater site that receives a PREPARE message requests write locks to update the data in its local work-area. As soon as the updates have been performed, a PREPARED message is returned to the cohort. The cohort sends a PREPARED message to the master after it receives PREPARED messages from all its updaters. As soon as the master receives PREPARED messages from all its cohorts, it validates globally the transaction by sending COMMIT messages to all the cohorts.

The write phase begins when a cohort receives a COMMIT message. After a cohort writes on the database, it sends COMMIT messages to its updaters to update the database replicas.

To resolve data conflict, two mechanisms are proposed by MIRROR based on the following observation: "it is very expensive to abort a transaction when it is near to completion because all the resources consumed by the transaction are wasted" [25].

- Priority Blocking (PB) mechanism: any transaction that cannot acquire a lock is inserted in a *lock request queue* until the required lock is released. The queue is ordered by transaction priority.

- Priority Abort (PA) mechanism: if a transaction *T* has higher priority than *R* and *T* requests a lock already held by *R* then instead of aborting *R* in order to execute *T* as it is done usually in real-time systems ; *R* will resume its execution if *R* is executing in its later stages. In this case, *T* will not wait too long since *R* is ending its execution. On the other hand, if R is executing in its first stages then it is aborted in order to execute *T*.

The rule introduced by Xiong et al. in [25] allows to use PA if the lock holder has not reached yet a *demarcation point* and to use PB after the demarcation point is reached. The demarcation point corresponds:

- to the reception from the master of a PREPARE message for a subtransaction ST_{ij}, if the receiver is a participant site and
- to the acquisition of all the local write locks by the local subtransaction if the site is a replica updater.

4.2 RCCOS Protocol

Our protocol is similar to MIRROR protocol under normal (non-overload) conditions. But when the system is overloaded because on each site, the computation time of transactions set exceeds the time available on the processor, we have to favour the executions of the most

important transactions according to the application-transactions set. The favoured transactions are those which undergo less timing faults and which are less dropped. 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 {\it importance value}, as described in \cite{kai98} for distributed real-time systems.

4.2.1 Transaction Importance

We associate, to each global transaction, a positive integer that represents the importance value of the transaction according to the application-transactions set. The importance value is intrinsic to the application, so it can be fixed by the application developer. Each transaction submitted to the master is characterized by a deadline which defines its urgency and by a value which defines the importance of its execution, with respect to the other transactions of the real-time application. The importance (or criticality) of a transaction is not related to its deadline; thus two different transactions which have the same deadline may have different importance values.

We define the importance value of a ready-transactions queue as the importance value of the most important transaction of the queue. If readyQueue_s is the ready-transactions queue of a site s, then the importance value of the site denoted Imp_s is:

$Imp_s = \text{Imp}(readyQueue_s) = \text{Max} \{\text{Imp}(T_i)\}; \forall\ T_i \in readyQueue_s$.

4.2.2 RCCOS Protocol Principle

RCCOS protocol is applied to the distributed DBMS (Data Base Management System) model described in section 3. It is upstream of MIRROR protocol. When the master receives a transaction T_i to execute, it splits it into subtransactions. For each subtransaction, it tries to find a participant site. Since the database is replicated on many sites, in normal (non-overload) situation, for each subtransaction ST_{ij} any site among those that have replicas of the data items needed by ST_{ij} can be chosen to designate the participant site.

But when the system is overloaded, the first step is to use a placement policy based on the importance concept. In other words, before executing subtransactions we have to find for each subtransaction ST_{ij} the participant site that manipulates the least important transactions, among m_{STij} sites that are candidates to be the cohorts. Particularly, if the importance values of the m_{STij} sites, collected by the master, are equal to the importance value of the global transaction T_i then T_i is aborted by the master because the importance values must be distinct within a site so as to distinguish between transactions with high importance values from transactions with low importance values. The site that has the least important transactions is chosen to accept the execution of ST_{ij}, hence to become the cohort, while the remainding sites that have replicas of the database will be the updaters. Our choice is motivated by the fact that, in overload conditions, we have to carry out the most important transactions in the system: it is then better to submit a subtransaction to a site that handles transactions with low importance values than the reverse.

In order to find a possible cohort for a subtransaction ST_{ij}, the master broadcasts an INITIATE message as well as the global transaction T_i to all the sites that have replicas of the data items needed by ST_{ij}. When a site receives an INITIATE message, it sends its importance value. As soon as the master has collected the importance values of all the concerned sites, it declars as a cohort the site that has the lowest importance value by sending to it a COMPUTE message in order to execute ST_{ij}. The site that receives a COMPUTE message is considered as the cohort. The other sites that fail to be the cohorts will be the updaters. The cohort, in this case, has to stabilize the new ready-transactions queue. The ready queue contains current ready transactions as well as the subtransaction ST_{ij} added to the queue during the treatment of the COMPUTE

message. The stabilization process consists in, when the system is overloaded, privileging the transactions that have high importance values. The transactions that have low importance values are released from the queue and aborted one after the other until the processor laxity is 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 order to stabilize the ready queue, the transaction that will be aborted is chosen among transactions with low importance values while applying PA real-time mechanism introduced by MIRROR protocol. Hence, the overload is resorbed by a rejection policy based on removing transactions that have not reached their demarcation point and that have minimal importance values. A site considered as the cohort for a subtransaction can be, at the same time, the updater for another subtransaction. Therefore, the demarcation point for a subtransaction may correspond to the reception of PREPARE messages or to the acquisition of all the required write locks depending on whether the site is viewed as the cohort or as the updater for this subtransaction.

5. RCCOS Algorithm

The algorithm uses some variables and data structures. We describe them in the following subsections.

5.1 Notations

- T_i denotes a global transaction submitted to the master site. It is composed of k subtransactions denoted by $\{ST_{i1}, ST_{i2}, ..., ST_{ik}\}$ that will be executed on k participant sites.
- Imp_{Ti} is a positive integer value that denotes the importance value of T_i. The importance values of the subtransactions are equal to the importance value of the global transaction to which they belong. Hence, $Imp_{STij} = Imp_{Ti} \ \forall j \in \{1, k\}$.
- Imp_s is the importance value of a site s. It corresponds to the highest importance value of the ready transactions within s.
- $Dead_{Ti}$ denotes the absolute deadline of T_i. $Dead_{Ti}$ is also the deadline of the subtransactions of T_i.
- E_{STij} denotes a set of sites that have replicated data items needed by ST_{ij}. This set includes the cohort and the updaters.
- $MinImp_{STij}$ is the lowest importance value of E_{STij} and s_{min} is the site that has $MinImp_{STij}$ as the importance value.
- *readyQueue* is a list that contains ready transactions. Each element of the queue has the structure of *readyTrans* as described below. The transactions are sorted by increasing their absolute deadlines. That is, the highest-priority transaction has the nearest deadline.
- ready*T*rans is the structure of each element of the *readyQueue* (*subtrans*: the subtransaction concerned, *globalTrans*: the global transaction to which belongs *subtrans*, *absDead*: the deadline of *subtrans*, *remExecTime*: initialized to the total execution time of *subtrans* and is decreased during the execution to store the pending execution time of *subtrans*, *next*: the next element in the queue).
- *importanceQueue* contains ready transactions that are sorted by increasing their importance values, that is, the first transaction of the queue is the least important one. Each element of the queue has the structure of *impTrans* as described below.
- *impTrans* is the structure of each element of the *importanceQueue* (*subtrans*: the subtransaction concerned, *globalTrans*: the global transaction to which belongs *subtrans*, *Imp*: the importance value of *subtrans*, *next*: the next element in the queue).
- *currentReady* is the current ready transaction dealt with when reading *readyQueue* elements.

- *computTime* is an integer value that cumulates the pending execution time of treated transactions.
- *deleteTime* is an integer value that represents the pending execution time of the transactions that will be aborted in order to alleviate the processor load.
- *nbResp* is an integer value that counts the number of messages not received yet.

5.2 Exchanged messages

- INITIATE(ST_{ij}, T_i) is a message sent by the master to find the site that manipulates the least important transactions, in order to be the cohort.
- IMP(ST_{ij}, T_i, Imp_s) is a message sent by a site s to inform the master about its current importance value.
- COMPUTE(ST_{ij}, T_i, Imp_{Ti}, $Dead_{Ti}$): this message is sent by the master to the cohort in order to trigger the execution of ST_{ij} on the cohort.
- ABORT(ST_{ij}, T_i): if the stabilization mechanism chooses ST_{ij} as a victim, an ABORT message is sent to the master that has to abort T_i after being sent ABORT messages to the other cohorts. The cohorts, of course, propagate this message to the updaters if the updates have been prepared. We note that no message ABORT is sent by the cohort if ST_{ij} misses its deadline because the deadline expiration is detected locally on the master, on the cohorts and on the updaters.

5.3 Text of the Algorithm

In order to designate a cohort for a subtransaction ST_{ij}, i.e., a site that has the lowest importance value in order to execute ST_{ij} of a global transaction T_i, the master broadcasts INITIATE messages to all the sites that have replicas of data items needed by ST_{ij}. Figure 2 summarizes the steps of the algorithm by a diagram and the details of the algorithm are given below. We first describe the behaviour of the master then that of the cohort.

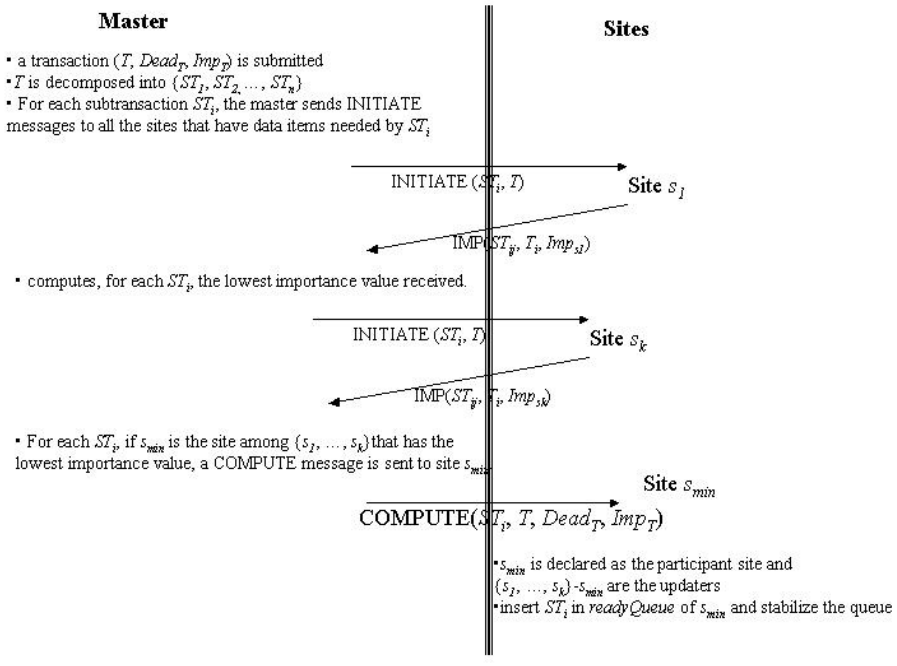

Figure 2. *The steps followed to place a subtransaction on a cohort*

Within the master

WHEN a transaction T_i composed of k subtransactions is submitted to the master:
DO
$\forall\, j \in \{1, k\}$:
BEGIN
$nbResp_{STij}$ = Cardinal (E_{STij});
$MinImp_{STij}$=+∞;
$\forall\, s \in E_{STij}$:
send INITIATE(ST_{ij}, T_i); /* broadcasts INITIATE messages to all the replicas sites */
END
END DO

WHEN the master receives IMP(ST_{ij}, T_i, Imp_s) from a site s:
DO
$nbResp_{STij} = nbResp_{STij} - 1$;
1 IF ($Imp_s \neq Imp_{Ti}$) AND ($Imp_s < MinImp_{STij}$)
THEN
$MinImp_{STij} = Imp_s$;
$s_{min} = s$;
2 ENDIF
IF ($nbResp_{STij} = 0$)
THEN
/* all the IMP messages have been received */
send COMPUTE(ST_{ij}, T_i, $Dead_{Ti}$, Imp_{Ti}) to s_{min};
/* s_{min} is the cohort of ST_{ij} */
ENDIF
END DO

Within any site:

WHEN a site s receives INITIATE(ST_{ij}, T_i) from the master:
DO
/* returns its importance value */
Imp_s = Max $\{Imp(ST_k)\backslash\}$ $\forall\, ST_k \in readyQueue_s$;
send IMP(ST_{ij}, T_i, Imp_s) to the master;
END DO

Within the cohort:

WHEN the cohort receives COMPUTE(ST_{ij}, T_i, $Dead_{Ti}$, Imp_{Ti}) from the master:
DO
/* stabilizes the local ready queue */
t=current-time();
insert (ST_{ij},T_i, $Dead_{Ti}$) in *readyQueue*;
insert (ST_{ij},T_i,Imp_{Ti}) in *importanceQueue*;
ov= OVERLOAD-PROCEDURE(t);
/* *ov* stores the overload value returned */
IF (ov ≠ 0)
THEN

```
            deleteTime = 0;
        3   trans=first transaction of importanceQueue;
            WHILE (trans ≠ null)
            DO
            4     IF (trans has not reached its demarcation point) THEN
                        release trans from importanceQueue;
                        release trans from readyQueue;
                        abort trans;
                        send ABORT(trans.subtrans,trans.globalTrans) to the master ;
                        deleteTime=deleteTime+trans.remExecTime;
                        IF (deleteTime ≥ ov) THEN exit;          ENDIF
                  ENDIF
                  trans = trans.next;
            5  END WHILE
      ENDIF
END DO

OVERLOAD-PROCEDURE (integer t)
BEGIN
      int loadValue=0;
      int computTime=0;
      currentReady= first transaction of readyQueue;
      WHILE (currentReady != null)
      DO
      /* if the laxity condition of currentReady is a negative value */
            IF (computTime +  currentReady.remExecTime ≥ currentReady.absDead - t)
            THEN  /* memorize the overload value */
                  loadValue = loadValue + currentReady.remExecTime;
            ELSE computTime = computTime + currentReady.remExecTime;
            ENDIF
            currentReady = currentReady.next;
      END WHILE
      return(loadValue);
END
```

6. Elements of proof

6.1 Definitions

6.1.1 Definition 1

A subtransaction R within a participant site s is defined thanks to four temporal parameters:

- r : the time R is available within s,
- C: the total computation duration,
- *Dead*: the absolute deadline inherited from the global transaction of R and
- *Imp*: its importance value.

We note that $Dead_R$-r_R is the relative deadline of R. $C_R(t)$ denotes the computation time that remains at time t to finish the execution of R.

6.1.2 Definition 2

Let $readyQueue_s$ be the list of ready transactions of a participant site s sorted at time t by increasing deadline:

$$readyQueue_{s,t} = \{R_0, R_1, ..., R_n\} \text{ where } C_{Ri}(t)>0 \ \ (\forall\ i \in \{1, n\})$$

We define the processor laxity LP as the conditional laxity LC of each transaction of $readyQueue_{s,t}$:
$LP(t)$= laxity($readyQueue_{s,t}$)= Min $\{LC_{Ri}(t)\}$ ($\forall$l $R_i \in readyQueue_{s,t}$) where $LC_{Ri}(t)= Dead_{Ri} - t - \sum_{j=0,\ i} C_{Rj}(t)$. The sum in j computes the pending execution time of all the transactions (including R_i) that are triggered at time t and that are preceding R_i in the assignement sequence.
An overload situation is detected as soon as the system 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, $|LP(t)|$.

- $LP(t) >0 \Rightarrow Dead_{Ri} > t + \sum_{j=0,\ i} C_{Rj}(t)$ ($\forall\ R_i$) means that: while allowing the execution of (i-1) transactions that precede R_i in the queue, R_i will finish its execution before the deadline expires.

- $LP(t) < 0 \Rightarrow (\exists\ R_k)\ Dead_{Rk} < t + \sum_{j=0,\ k} C_{Rj}(t)$ ($\forall\ R_i$) means that, at least, we have one ready transaction R_k for which the deadline expires before R_k ends its execution.

6.1.3 Definition 3

$readyQueue_{s,t}^{stable}$ is a stabilization of $readyQueue_{s,t}$ if:

- $readyQueue_{s,t}^{stable} \subseteq readyQueue_{s,t}$,
- laxity($readyQueue_{s,t}^{stable}$) ≥ 0 and
- if $\Re = readyQueue_{s,t} - readyQueue_{s,t}^{stable}$ then ($\forall\ R \in \Re$) ($\forall\ \Im \subseteq readyQueue_{s,t}^{stable}$) we have: ($\forall\ R' \in \Im$, $Imp\{R'\} < Imp_R$) $\Rightarrow$ laxity($readyQueue_{s,t}^{stable} - \Im) \cup \{ R\}$) < 0.

The stabilization process consists in privileging, when the system is overloaded, transactions with high importance values. For this purpose, we remove from $readyQueue_{s,t}$ transactions with low importance values until the laxity is positive. This is expressed in point 2. The third point means that we should not remove a transaction R from $readyQueue_{s,t}$ so that the new list contains transactions that have lower importance values than R ; moreover if we remove theses transactions from the queue we could reinsert R in the queue while maintaining a positive laxity.

6.2 Text of the proof

Theorem 1 *If a global transaction T_i is composed of n subtransactions, $T_i = \{ST_{i1}, ST_{i2}, ..., ST_{in}\}$, then for each subtransaction ST_{ij} with importance value $Imp\{ST_{ij}\}$ we have at most one site that accepts the execution of ST_{ij}.*

Assume that we have two distinct sites s_0 and s_1 that accept ST_{ij}. In this case, $MinImp_{STij}$ that is different from Imp_{STij} (see line 1 of the algorithm) stores the minimum importance value of a site. If s_0 and s_1 are selected to execute ST_{ij}, this means that s_0 and s_1 have $MinImp_{STij}$ as importance value, that is, $MinImp_{STij}=Imp_{s0} = Imp_{s1}$. But while calculating $MinImp_{STij}$ (see line 1 to line 2 of the algorithm), if many sites have the same importance value, s_{min} stores only the identity of the first site that sends its importance value. $MinImp_{STij}$ will not be updated until a site sends a lower importance value. Hence, we cannot have two participant sites for the same subtransaction.

Theorem 2 *Within a participant site s, a subtransaction ST_{ij} that has Imp_{STij} lower than Imp_s and that reaches its demarcation point is not aborted in overload conditions unless its deadline expires.*

Theorem 2 says that, when the processor is overloaded, ST_{ij} is not aborted if it has reached its demarcation point and remains in the stable queue. Assume that at time t, $ST_{ij} \notin \text{readyQueue}_{s,t}^{\text{stable}}) \Rightarrow \text{laxity}(readyQueue_{s,t}^{\text{stable}} \cup \{ST_{ij}\}) < 0$. Moreover, since $Imp_{STij} < Imp_s$, ST_{ij} should be removed from $readyQueue_{s,t}$. But according to line 4 of the algorithm, ST_{ij} is not aborted since it has reached its demarcation point. Hence $ST_{ij} \in readyQueue_{s,t}^{\text{stable}}$.

Theorem 3 *Within a participant site s, if there are transactions with low importance values, that have not reached their demarcation point and whose pending execution time is greater than the overload value then the most important transaction meets its deadline.*

Let R be the most important transaction in $readyQueue_{s,t}$, that is, $Imp_R=Imp_s$. If the processor is overloaded, this means that at time t:

$$(\exists\ \Im \in readyQueue_{s,t})\ (\forall\ T \in \Im)\ Imp_T < Imp_s$$

so that T has not reached its demarcation point and $\text{laxity}(readyQueue_{s,t})<0$ and $\sum_{Ti\in\Im} C_{Ti}(t) > t + Dead_{Ti}$. But according to line 3 to line 5 of the algorithm, because of their low importance values, all the transactions of $\Im$ will be aborted one by one until the laxity of $readyQueue_{s,t}$ becomes positive, that is:
$\sum_{Ti\in\Im} C_{Ti}(t) = 0 \Rightarrow \text{laxity}(readyQueue_{s,t} - \Im) > 0 \Rightarrow readyQueue_{s,t} - \Im = readyQueue_{s,t}^{\text{stable}} \Rightarrow R \in readyQueue_{s,t}^{\text{stable}} \Rightarrow R$ will meet its deadline.

7. Experimental platform

As it is shown in Figure 3, transactions are submitted via Web pages to the master which triggers subtransactions on different replica sites while handling the sites overload. On each site, there are three modules:
- the *scheduler* module that uses EDF algorithm to schedule local subtransactions,
- the *replicas data manager* that resolves data conflict and that handles replicated data and
- the *overload controller* module that implements RCCOS protocol.

This platform has been developped in Java and JDBC/ORACLE. The first experiments that have been done show that among transactions that meet their deadlines, the majority concerns transactions with high importance values. This behaviour can be explained as follows: the great percentage of deadline meeting is due to transactions with high importance values because RCCOS favours this kind of transactions. The little percentage of transactions with low importance values that meet their deadlines is due to the transactions that have reached their demarcation point, hence they are not aborted when applying stabilization process.

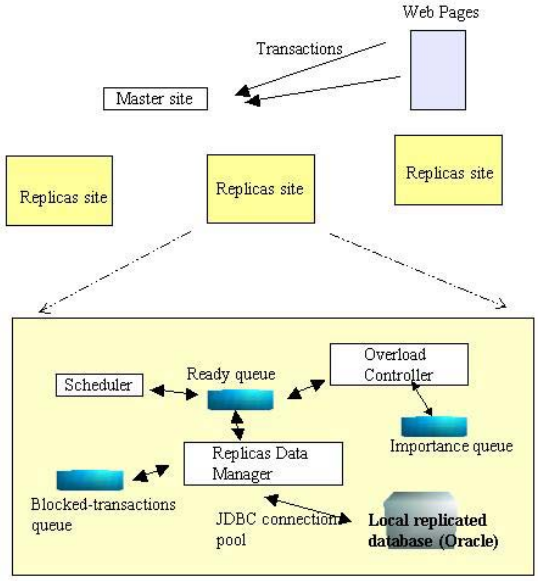

Figure 3. *The different modules of the experimental platform*

8. Conclusion and future work

In this paper, we have addressed the problem of managing firm-deadlines transactions that access replicated data in a distributed system that may be overloaded. For this environment in which many current time-critical applications operate, specially Web-based services, we proposed a novel protocol called RCCOS. This protocol can be easily integrated in current systems to handle processors overload without altering the database consistency which is the main objective of DRTDBSs. The main idea of the protocol is to associate an importance value with each submitted transaction in order to favour, when the system is overloaded, the executions of the most important transactions according to the application-transactions set. In overload conditions, each subtransaction of the global transaction is executed on a site that has the lowest importance value among those that have replicas of the data items needed by the subtransaction. Moreover, the participant site has to stabilize its ready queue, that is, it has to manage the processor overload by maintaining the execution of the most important transactions and by aborting the others. Priority Abort real-time mechanism of MIRROR protocol has been added to RCCOS protocol when proceeding to stabilization so as to favour the deadline meeting for the transactions that have high importance values and/or that are near to completion. The first experiments that have been done on our platform show that the majority of transactions that meet their deadlines have high importance values. We are currently working on integrating MIRROR protocol into our simulation platform to investigate the performance improvements that may arise from RCCOS protocol when comparing the performance of MIRROR and RCCOS protocols. We will also investigate mechanisms to minimize communications and response times between the cohort and the updaters when managing real-time database replicas.

Aknowledgement
Special thanks to SAAD Abdelkrim who helps us in the development of the experimental platform.

References

[1] A. Atlas and A. Bestavros, "Statistical Rate Monotonic Scheduling", in *proc. of IEEE Real-Time Systems Symposium*, Madrid, dec. 1998.
[2] T. Anderson, Y. Breitbart, H. F. Korth and A. Wool, "Replication, consistency and practically: Are these mutually exclusive?", *in Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 484-495, Seattle, WA, USA, June 1998.
[3] D. Agrawal, A. El Abbadi and R. C. Steinke, "Epidemic Algorithms in Replicated Databases (extended abstract)", *in Proc. of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Princicples of Databases Systems*, pp. 161-172, Arizona, USA, May 1997.
[4] S. K. Baruah and J. R. Haritsa, "Scheduling for Overload in Real-Time Systems", *IEEE Trans. on Computers*, **46(9)**, pp. 1034-1039, sept. 1997.
[5] P. Bernstein, V. Hadzilacos and N. Goodman, "Concurrency Control and Recovery in Database Systems", *Addison Wesley*, Massachussets, 1987.
[6] 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.
[7] M. Carey and M. Livny, "Conflict Detection Tradeoffs for Replicated Data", *ACM Transactions on Database Systems*, **Vol. 16**, pp. 703-746, 1991.
[8] F. Cottet, J. Delacroix, C. Kaiser and Z. Mammeri, "Scheduling in Real-Time Systems", *Wiley Edition*, Hardcover, nov. 2002.
[9] J. Delacroix, "Towards a Stable Earliest Deadline Scheduling Algorithm", *Real-Time Systems Journal*, **10(3)**, pp. 236-291, may 1996.
[10] J. Delacroix and Christophe Ménival "Intégration d'un Contrôle de Charge par Importance au sein du Système RT-Linux", *RTS'2000 Conference*, pp. 47-63, march 2000, Paris.
[11] J. Hansson, S. H. Son, J. A. Stankovic and S. F. Adler, "Dynamic transaction scheduling and reallocation in overloaded real-time database systems", *In Proc. of the 5th Conference on Real-Time Computing Systems and Applications}*, pp.293-302, IEEE Computer Press, 1998.
[12] J. Hansson and Sang H. Son, " Real-Time Database Systems : Architecture and Techniques", K. Lam and T. Kuo (eds.), Kluwer Academic Publishers, pp. 125-140, 2001.
[13] J. Haritsa, M. J. Carey and M. Livny,"Earliest Deadline Scheduling for Real-Time Database Systems", *Proc. of 1991 IEEE Real-Time Systems Symp.*,dec. 1991.
[14] H. Huang, J. A. Stankovic, K. Ramamritham, D. Towsley and B. Purimetla, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proc. of the 17th International Conference on Very Large Data Bases*, Barcelona, Sept. 1991.
[15] C. Kaiser, C. Santellani, "Pétrarque. Une Plate-forme d'Expérimentation pour l'ordonnancement temps réel strict d'applications réparties", *Technique et Science Informatique Journal*, **17(1)**, pp.39-62, 1998.
[16] 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.
[17] B. Kemme and G. Alonso, "A suite of database replication protocols based on group communication primitives", *in Proc. of the International Conference on Distributed Computing Systems*, pp. 156-163, Amsterdam, May 1998.

[18] C. Liu and J. Leyland, "Scheduling Algorithms for Multiprogrammig in Hard Real-Time Environment", *Journal of the ACM*, **20(1)**, 1973.
[19] A. K. Mok and D. Chen, "A multiframe model for real-time tasks", *IEEE transactions on Software Engineering*, **23(10)**, p. 635-645, 1997.
[20] F. Pedone, R. Guerraoui and A. Schiper, "Exploiting atomic broadcast in replicated databases", *in Proc. of the 4th International Euro-PAr'98 Conference*, **Lecture Notes in Computer Science 1470**, pp. 514-520, Southampton, sept. 1998.
[21] S. Son, "Using Replication for High Performance Database Support in Distributed Real-Time Systems", *Proceedings of the 8th IEEE Real-Time Systems Symposium*, pp. 79-86, 1987.
[22] S. Son and S. Kouloumbis, "A Real-Time Synchronization Scheme for Replicated Data in Distributed Database System", *Information Systems*, **18(6)**, 1993.
[23] S. Son and F. Zhang, "Real-Time Replication Control for Distributed Database Systems: Algorithms and Their Performances", *4th International Conference on Database Systems for Advanced Applications*, Singapore, April, 1995.
[24] O. Ulusoy, "Processing Real-Time Transactions in a Replicated Database System", *Distributed and Parallel Databases*, **2**, pp. 405-436, 1994.
[25] M. Xiong, K. Ramamritham, J. Haritsa and J. A. Stankovic, "MIRROR: A State-Conscious Concurrency Control Protocol for Replicated Real-Time Databases", *Information systems*, Elsevier Science Publishers, **27(4)**, pp. 277-297, june 2002.