Deadlock Free Concurency Control by Value Dates for Scalable Distributed Data Structures

Ibrahima E. Kane[1], Witold Litwin[2] Tidiane SECK[1], & Samba Ndiaye[3]

Abstract :

We define a concurrency control scheme for a Scalable Distributed Data Structure. The scheme avoids deadlocks, including the distributed ones, through the use of transactions with value dates. Comparison of value dates allows to abort and later restart selected transactions among those that conflict and could deadlock. We present our schema, its implementation and experimental performance analysis. We show that the scheme should offer satisfactory performance for the practical.

1. Introduction

A Scalable Distributed Data Structure (SDDS) stores application data in a file transparently distributed over the nodes of a multicomputer, [LNS93]. The file consists of records identified each by a *primary* or a k-d key. Each storage node, the *server* node of the SDDS, stores the record in a *bucket*. The number of storage nodes, dynamically scales with the file size through the splits of the overloaded buckets. A split typically evacuates half of a bucket to a new bucket at a new server appended to the SDDS.

The application interfaces only the SDDS *client* component on its node. The record address calculus at the client from the key value does not require access to any central repository. This could otherwise constitute a hot spot. The client uses its *image* of the file structure. The image can be inaccurate, as SDDS splits are not posted to the clients (clients can be unavailable when spits occur, or mobile…). The client may send the key search or the insert with the record to an incorrect server, or may simply invokes a multicast message. Each SDDS server has therefore a built-in algorithm to check whether its address is the correct one for the incoming query. If not, and the query is a unicast message, it forwards it to a server which could be the correct one. The new server iterates the same procedure, as it may still be the incorrect one. Ultimately, possibly only after a few hops, the query reaches the correct server. The client gets then the *Image Adjustment Message* (IAM). The IAM content allows the client to adjust its image, at least so that the same error does not occur twice.

Many SDDS schemes are now known. The LH* schemes for the scalable distributed hash partitioning, and the RP* schemes for the scalable distributed range partitioning were the most studied. It was shown that they provide excellent scalability. In practice, only the number of available nodes and their storage limit the file size. The LH* or RP* file can thus potentially scale to the magnitudes impossible in practice for more traditional data structures.

The SDDS-2000 prototype offers these schemes for the data storage in the distributed RAM of a *network multicomputer*, i.e., a high-speed network of popular computers. Its performance analysis has shown its capability to handle million-record SDDS files with key search or record insert times of 0.1 – 0.3 ms. These times are potentially a hundred time faster than those to traditional disk files.

[1] Ecole Nationale Supérieure Polytechnique, U. Dakar
[2] U . Paris 9 Dauphine
[3] Dep. Mathématiques & Informatique, U. Dakar

An SDSS file shared by several applications needs a concurrency control. A deadlock free scheme is preferable, as usual in a distributed system. The concept of *value date* allows for such schemes, [WLH88]. A value date *V* for a transaction *T* is a time limit assigned so that *T* must terminate at most by *V*. Every transaction should get a different value date. If two transactions conflict, their value dates can be compared. If the comparison shows that a deadlock could occur, one of the transactions is aborted and restarted later with a new *V*, chosen to avoid the conflict. This is the principle of the VDAS schema in [WLH88] and [WLH89].

Two cases of SDDS data sharing appear in practice. First, a file can be private to the applications at the same client only. We refer to it as a *single* (*SDDS) client* case. More generally, a file can be shared by multiple SDDS clients. This is the *multiple* (*SDDS) client* case.

The single client case allows for the concurrency and transaction management only at the client. The multiple clients require the concurrency control also at the servers. The former property is attractive, as it allows for the coupling with any known centralized scheme. However the performance of the coupling have to be determined. The overhead of the any concurrency management scheme affects the performance of the SDDS, perhaps unacceptably.

Below, we present the concurrency manager for the single client case, coupling VDAS scheme with SDDS-2000. The system required the study of various issues. One is the strategy for the calculus of the value dates for the VDAS scheme. This choice influences the ratio of the restarted transactions and the resulting overhead. Some transactions may in particular potentially restart several times. This creates the potential for the livelock that we should prevent. The whole overhead depends on the load of the system, on the length of the transactions involved etc. We thus had to find out what performance, especially the throughput, our implementation could finally offer in practice.

We first describe our design choices. We recall the VDAS scheme and show how we completed it to avoid the livelock. We then present our method for the value date determination. Next, we analyze the system performance. For this purpose, we carry the simulations of various transactions entering our manager. The results prove the effectiveness of our scheme. Only a fraction of transactions restarts, and a few times only, leading to an efficient throughput.

Our results are of importance beyond their application to an SDDS. Under the name of *transactions with deadlines*, value dates have been extensively studied for real-time databases. We are aware of theoretical analysis only. Our scheme is the first implemented to the best of our knowledge.

Section 2 presents our concurrency management. Section 3 discusses the performance. Section 4 concludes the study.

2. Concurrency management

2.1 Basic VDAS scheme

A *VDAS-transaction* is basically any ACID transaction provided with the value date. Other non-atomic transaction models can be applied as well, e.g., the flexible transaction model [_]. The value date is computed by the application, or the transaction manager. Obviously, it has to be far enough to allow the transaction to complete. How it is computed however is not the part of the VDAS schema. The only condition is that no two concurrent VDAS-transactions ever enter with the same value date.

To manipulate (read or write) a data item a VDAS-transaction *T* has to stamp it with its value date. The data that could be stamped by *T* behaves as it was locked by *T* in the usual sense, until T terminates with a commit or abort. The lock can be considered exclusive or shared. It is granted to T if the data item does not already carry another lock (value date).

In the latter case two VDAS-transactions *conflict*. Let D be the item, let T_1 be the transaction with value date V_1 that already locked D, and let T_2 with V_2 be the one that requests D. The VDAS conflict resolution rule is as follows:

(1) if $V_2 > V_1$, T_2 waits else abort and restart either T_1 or T_2 with new V.

The deadlock avoidance is proven in [WLH88]. In short, T_1 and T_2 never deadlock since only T_2 can wait. The choice of the abort and restart victim, as well as of its new value date are not the parts of the VDAS scheme. These are nevertheless very important choices. A naïve approach, such as "always abort T_1" may lead to a livelock. If V depends essentially on the duration of T, and T_1 is a very long transaction while most of others are very short, and come randomly with the rate much shorter than T_1 duration, T_1 may end up being aborted systematically.

The waiting time imposed by rule (1) to T_2 may also potentially cause it to reach V_2 without the completion and thus get aborted anyhow. T_2 restarts then with a new value date $V_2' > V_2$. The restart clearly does not guarantee the completion by V_2'. Notice finally that transactions with value dates cannot deadlock even without rule (1). Any interlock lasts only until the smallest value date of the transactions involved in it.

The transaction management is beyond the scope of VDAS scheme and of our manager we report on. Notice nevertheless that VDAS scheme behaves for ACID transactions as the most popular strict two-phase locking (2-PL) schema. It thus guarantees the serializability of the VDAS-transactions.

In our case, the granularity of locking is basically an SDDS record. However, VDAS scheme works for any granularity. In our single-client case, one-phase commit (1 PC) suffices. In general, one can apply also other popular commit protocols, e.g. the 2 PC. For transaction schemes beyond ACID, e.g., for the flexible transactions, VDAS allows for new types of commit. The *implicit* commit by value date is especially promising. The commit process does not require any specific message from the coordinator, if no server requests an abort before the value date. This is an important advantage over 2 PC for a larger number of participants, e.g., SDDS servers in the multiple client case.

2.2 Priority based VDAS schema

To avoid the livelock and more generally multiple restarts of any transaction, we have completed the basic VDAS scheme with the *priority* management. In our priority based VDAS scheme, every VDAS transaction T gets an identifier I when the application submits it. T keeps I when it restarts, until it completes or the application drops it. Every T also have some priority P ; $0 \le P \le \boldsymbol{P}$. Every new T gets $P = 0$. Then P increases by one with every restart of T, until some maximal $\boldsymbol{P}$. Let again T_1 and T_2 conflict with current priorities P_1 and P_2 both smaller than $\boldsymbol{P}$. Let also $\underline{P}$ be a parameter ; $0 < \underline{P} < \boldsymbol{P}$. The new conflict resolution is given by rule (2) below. It means that basic VDAS applies until one of the priorities reaches $\underline{P}$. Then, the resolution starts to be based on P value. We used rule (2) for our prototype.

```
(2)   If $P_1 = P_2$ or max $(P_1, P_2) < \underline{P}$ then          /* Case of rule (1)
          if $V_2 > V_1$ then $T_2$ waits else abort $T_1$
      Else if $P_1 < P_2$ then abort $T_1$
      Else if $V_2 > V_1$ then $T_2$ waits else abort $T_2$.      /* Case of $P_2 < P_1$
```

Any transaction T, whose P reaches $\boldsymbol{P}$ gets longest possible value date and goes to a special first-in first-out queue Q. The transactions in Q execute serially. They thus cannot conflict with each other, while they abort or put to wait through (2) any transaction they conflict with. No lifelock may thus occur. Notice however that the Q length may increase arbitrarily if the system can't accommodate the throughput.

Notice that when the transaction to abort to solve the conflict can be either one, it may be be less costly to choose the one that has less operations to redo. Our priority rules above do not take to the account this criterion. They can nevertheless be easily be expanded to do so, using as basis the data for calculus of the value dates that follows.

2.3 Value date calculus

The transaction manager knows for each transaction the estimate of the number of reads and writes it should perform. This description is supposed provided by the application or inferred by the transaction manager. The concurrency control manager also dispose of the estimates of time to complete an operation. Let n_R and n_W be respectively the number of reads and writes, t_R and t_W times to read or write, and $\varepsilon > 0$ a real number. The factor ε is a provision for estimation error and for the waiting time during the execution. The manager estimates the length L_m of the transaction to (re)start for m-th time as :

$$L_0 := (n_R * t_R + n_W * t_W) * (1 + \varepsilon_0)$$

$$\varepsilon_m := 2 * \varepsilon_{m-1}$$

$$L_m := L_{m-1} * (1 + \varepsilon_m)$$

In other words, ε doubles at each restart. Increasing ε_m is necessary to prevent a permanently incorrect estimate, hence a livelock. The initial ε_0 value should be generous enough so that T_2 waiting for the 1st time does not have V expired before it could finish. For the experimental performance analysis, we set ε_0 simply to 1. The reason is that (i) transactions generated for our performance study have the same number of operation and that (ii) every operation of a transaction is equally likely to generate a conflict, hence to abort T_2.

The value date V is computed as $V = L_m + D_m$, where D is the current time (date-time) when first operation of T is launched by the manager for the m-th execution. The operation use the usual services of SDDS-2000 [ref].

3. Performance measurements

To validate our manager, we have studied its performance through the simulation of transactions operating over the actual SDDS-2000 system. The simulation consisted in the generation of *streams* which are series of concurrent transactions. The stream *size* which is the number of transactions per stream was a parameter we have varied. The concurrency resulted from the multithreaded launches of the streams prepared for launch in a queue. The transactions conflicted randomly on predefined set of a thousand records in an RP* file. For the file access, our manager used an RP*$_N$ client. We had to group several SDDS servers of this file at the same machine, as we disposed of a few machines only. Hence the CPU bandwith of each machine was shared among the servers. This slowed the response times accordingly; with respect to the normal SDDS case of one server per machine.

First, as the reference, we have determined performance of the system for a single transaction in the system whose number of operations was a parameter. Next, we have generated multiple conflicting transactions. Of prime interest to the efficiency of our manager was the quantity of restarts, and especially of multiple restarts. Also, we had to know the incidence of the concurrency management on the execution time of a transaction, with respect to its execution time determined when the transaction was alone in the system. then analysed the behaviour of the concurrent transactions We have expected that the system will respond somehow differently have sknow the One aspect was the correctness The characteristic of main interest was the efficiency of the conflict resolution. Especially, the ratio of aborts and of multiple restarts. rate behaviour of In this environment, we have studied the timeliness and scalability of the entire system. Since our goal was the concurreOur objective was to validate the transaction manager by studying its behaviour when a certain number of the transaction model key parameters need to be evaluated :

- the transactions length (number of operations)
- competition degree (transactions batch size).

The indicators retained to qualify the transaction manager behaviour are :

* the conflict rate (abortion rate and locking rate)
* number of other attempts
* the average time of execution of a transaction

- the transaction manager stability study in continuous running of the transaction manager.

6.1- <u>Test environment</u>

The measurements were made on three machines having identical configurations, connected to a local network of 10Mb/s. One of the machines is used to serve as a client and the two others support.
The 10 SDDS servers on the basic of 5 servers a machine.
The common characteristics of the sets used for the test re as follow.

- operating system : windows N T server
- CPU type : Pentium II 400 MHz
- RAM : 64 MB

All measures to be carried out are related to the client (transactions manager).
We specify that the response time of the servers involved in the transactions operation is included in the measures made.

6.2- <u>A transaction execution time variation in function of its length</u>

We have measured the total execution time per transaction and per operation. The ideal scalability would mean that the total time is proportional to the transaction's length, and that the time per operation is constant. We expected some deviation of this ideal.
In this part is studied the variation of the execution time of a transaction of its length, that is the number of operations it generates, and this in succession for a reading and updating transaction.
This approach will not only allow the execution time evolution of a transaction to be observed in function of its length but will also make it possible to bring out the processing cost of an updating transaction compared with the one executing only reading operations.
To this end, measurements of a transaction response time will be made by gradually increasing transaction. In this study, the keys manipulated by the transaction different operations are sequentially generated. i.e. that to leave of a transaction length 100 the requests of redirection will be carried out by first server SDDS [AWD98].

The following graph is deduced from taken measurements. It presents the variations of execution time of a transaction of reading on the one hand and a transaction of update on the other hand according to its length

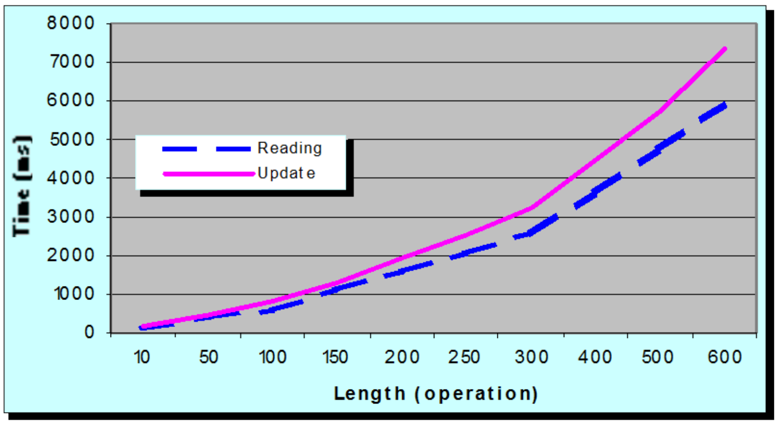

Fig6.1. *Variation of the execution time of a transaction according to its length*

This graph (Fig6.1) shows that execution time of a transaction does not vary in an exponential way when one varies his length (operation number). This result joined the hypothese of the SDDS scalability.

From these same measurements, we deduced the graph below. It shows the execution time variations by reading operation on the one hand and of updating on the transaction.

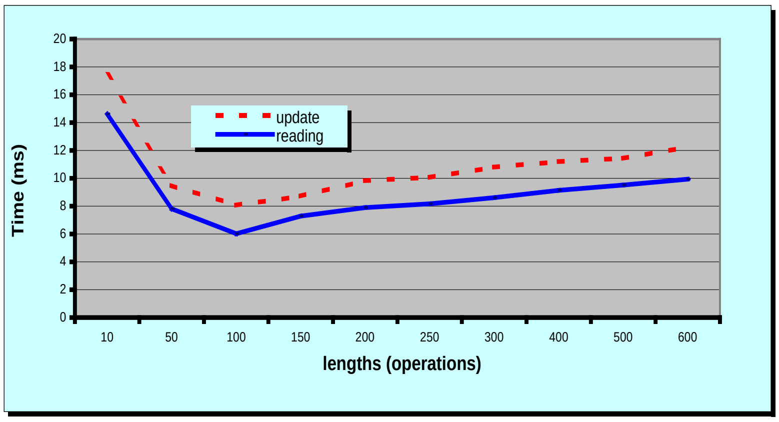

Fig 6.2. *time by operation in function of the transaction length.*

On the graph (*Fig 6.2*), a decrease in the execution time per transaction operation can be noted up to 100 operation, and then the beginning of a slight increase for a reading transaction as well as an updating transaction.

In spite of that slight gradual rise of the execution time per operation, the times values per operation for a 600-operation transaction (9.94 ms a reading operation and 12026 ms an up-dating operation) are lower than those of a 10 – operation transaction (14.6 ms a reading operation and 17.44 ms an up – dating operation).

The time decrease per operation can be explained by the client and before servers speeds gradual increase. Form a length transaction superior to 100 operations, the first server operates a request redirection in multicast towards the nine other servers. The combined effect of this phenomenon and the client has reached its maximum speed can explain the slight time rise per operation starting a transaction lengthier than 100 operations. On that account, the influential parameter on the execution time per operation with the client is its speed.

6.3. <u>Study of the transaction manager behaviour in a concurrent environment</u>.

The transaction manager behaviour in a competitive environment was studied in this part. The VDAS algorithm and the priorities based extra module behaviours were distinguished in this study.

The experimental procedure consisted in starting series of updating transactions steam of the same lengths 10 operations), and gradually changing the stream size. The transactions recording keys were randomly generated on a key area between 1 and 1000.

The two measurements were made in this part. In the first was recorded the transactions different streams execution time as well as the conflicts numbers (and locks) observed. The results are recapitulated in the table below.

Stream size (transactions)		**2**	**4**	**6**	**8**	**10**	**50**	**100**	**200**	**300**	**400**
Times (ms)		**401**	**762**	**1001**	**1512**	**1432**	**7901**	**19295**	**39574**	**58083**	**78643**
Abortion number	**Vdas**	**0**	**0**	**3**	**2**	**4**	**20**	**40**	**76**	**140**	**194**
	Suppl	**0**	**0**	**0**	**2**	**0**	**9**	**54**	**124**	**149**	**248**
Lock number	**Vdas**	**1**	**2**	**0**	**3**	**2**	**21**	**48**	**92**	**320**	**403**
	Suppl	**0**	**0**	**0**	**5**	**7**	**28**	**53**	**64**	**226**	**305**
Conflicts number		**1**	**2**	**3**	**12**	**13**	**78**	**195**	**356**	**835**	**1150**

Tab6.1. *Measurement of the execution time Of a transactions stream, (of the number) of abortions and locks of simultaneous execution transactions*

Then the number of times each stream transaction was restarted in the system. As a matter of fact, each aborted transaction was placed ion a waiting list and then restarted later in the system. The stream execution came only to an end when al the stream transactions were committed. Those measurements results are recorded in the following table *(Tab 6.2)*

		Number of restarts					
		1	2	3	4	5	6
	2	**0**	**0**	**0**	**0**	**0**	**0**
	4	**0**	**0**	**0**	**0**	**0**	**0**

Stream sizes **(transactions)**	**6**	**3**	**0**	**0**	**0**	**0**	**0**
	8	**4**	**0**	**0**	**0**	**0**	**0**
	10	**4**	**0**	**0**	**0**	**0**	**0**
	50	**18**	**4**	**1**	**0**	**0**	**0**
	100	**37**	**19**	**5**	**1**	**0**	**0**
	200	**73**	**35**	**15**	**3**	**0**	**0**
	300	**82**	**42**	**26**	**8**	**2**	**1**
	400	**91**	**66**	**39**	**19**	**4**	**1**

Tab.6.2. *Number of times a transaction has been restarted.*

Form the table *Tab6.1* can be inferred the average execution duration per transaction in a stream by dividing the total execution time by the stream size. The abortions, jamming and therefore conflicts propositions are calculated comparatively to the conflicts total number. That evaluations are recorded in the following table *(Tab 6.3):*

Stream size (transactions)		**2**	**4**	**6**	**8**	**10**	**50**	**100**	**200**	**300**	**400**	**Global everage**
Exécution time per transaction (ms)		**200,500**	**190,500**	**166,800**	**189,000**	**143,200**	**158,020**	**192,950**	**197,870**	**193,610**	**196,608**	**182,909**
Ratio of abort	**Vdas**	**0,000**	**0,000**	**0,333**	**0,167**	**0,308**	**0,256**	**0,205**	**0,213**	**0,168**	**0,169**	**0,190**
	Suppl	**0,000**	**0,000**	**0,000**	**0,167**	**0,000**	**0,115**	**0,277**	**0,348**	**0,178**	**0,216**	**0,130**
Globals proportions of abortion		**0,000**	**0,000**	**0,333**	**0,333**	**0,308**	**0,372**	**,0482**	**0,562**	**0,346**	**0,384**	**0,320**
Locks proportions	**Vdas**	**1,000**	**1,000**	**0,000**	**0,250**	**0,154**	**0,269**	**0,246**	**0,180**	**0,383**	**0,350**	**0,383**
	Suppl	**0,000**	**0,000**	**0,667**	**0,417**	**0,538**	**0,359**	**0,272**	**0,180**	**0,271**	**0,265**	**0,297**
Globals Proportions of locks		**1,000**	**1,000**	**0,667**	**0,667**	**0,692**	**0,628**	**0,518**	**0,360**	**0,654**	**0,616**	**0,680**

Tab6.3. *Measurement of the average execution time by transaction in a concurrent environment, the abortions and locks proportion related to the conflicts.*

It can be inferred that a transaction average execution time is if the order of 182.909 milliseconds. This evaluation also shows 32 % of the conflicts (19 % due to the algorithm VDAS and 13 % to the extra module) lead to abortions and the rest (68) to jamming and 29.1% due to the extra module.

The average conflict rate (T_c) with which the transactions were executed is computed started from the following formula translating the different streams conflict rate average :

$$T_c = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{number\ of\ conflicts}{executed\ operations\ number}\right)_i$$

In practice, it appeared difficult to note the executed operations number in a stream after its commit. That is why the average conflict rate observed in the system will be framed by two values (T_{min} and T_{max}) for which

$T_{min} < T_c < T_{max}$

Will always be obtained, where

$$T_{min} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{number\ of\ conflicts}{(stream\ size + aborts\ number)*10}\right)_i$$

And

$$T_{max} = \frac{1}{N}\sum_{i=1}^{N}\left(\frac{number\ of\ conflicts}{stream\ size*10}\right)_i$$

It is later found the average conflict rate is included between 8.92 % and 15.24 %

$$8.92\ \% < T_c < 15.24\ \%$$

The variation curve of the conflicts number *(Fig6.3)* and that of the average execution time by transactions *(Fig6.4)* in function of a stream size are respectively represented in the following two graphs.

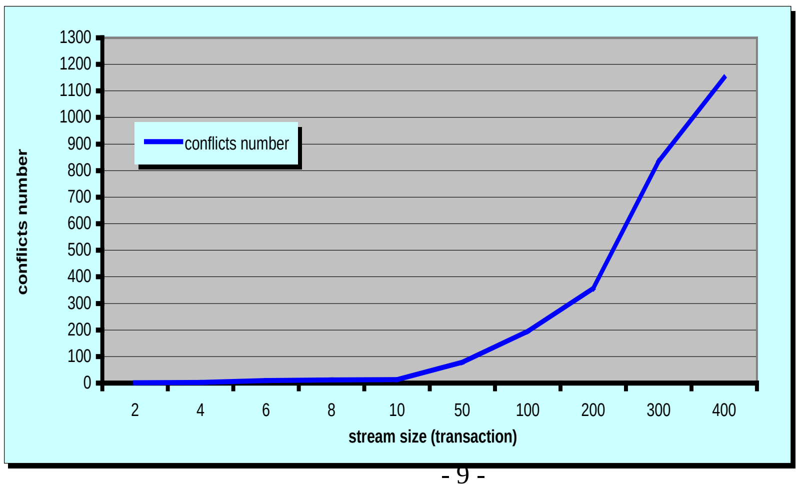

Fig 6.3 *conflicts number variation depending on a stream size*

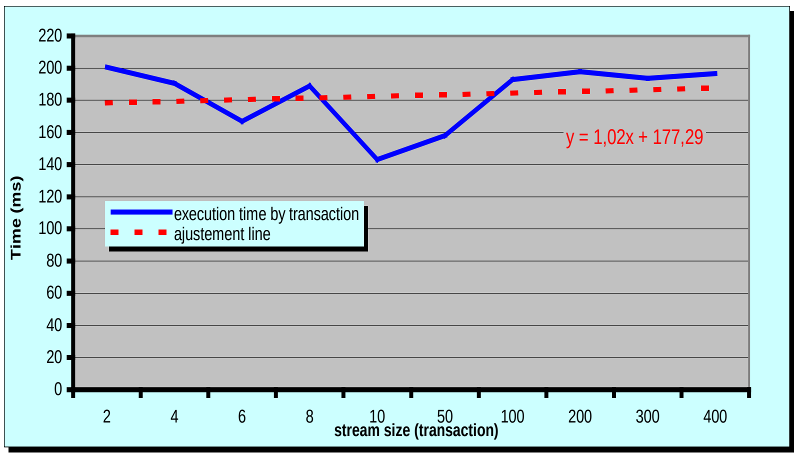

Fig 6.4. *A transaction execution time variation depending on a stream size*

The preceding graph *(Fig 6.4)* shows a transaction execution time does not exploited for transactions streams executed in an environment liable to a conflict rate between 8.92 % and 15.24 %. This result verifies the SDDS scalability hypothesis for our transaction manager.

6.4 <u>Study of the transaction manager stability</u>.

In real operation, the transactions from applications are placed in a waiting list the transaction manager manages in FIFO. In the same why, if a transaction aborts, it will be put in the waiting list and will, in this case, be considered as new transaction which will, at the right time, be restarted for operation in the system.

The object of this study is to determine a stability condition for the transaction waiting list. That is to determine a condition for which this list length is almost constant even when the transactions manager is set going for a long enough duration.

To this end, a stream of 10 length updating random transactions (10 operations) is started at each time interval D_t, for a time D. Then, the size L of the list is noted down at each time just before a new stream arrives.

The procedure is repeated until the list size variations between two consecutive streams are equal to zero for a given value D_t. This D_t value will in this case represent an acceptable value for the transactions list stability.

The following graph illustrates the aspects of three variation curves of the transactions list size, corresponding each to a D_t given value and for a D= 60.000 milliseconds execution duration.

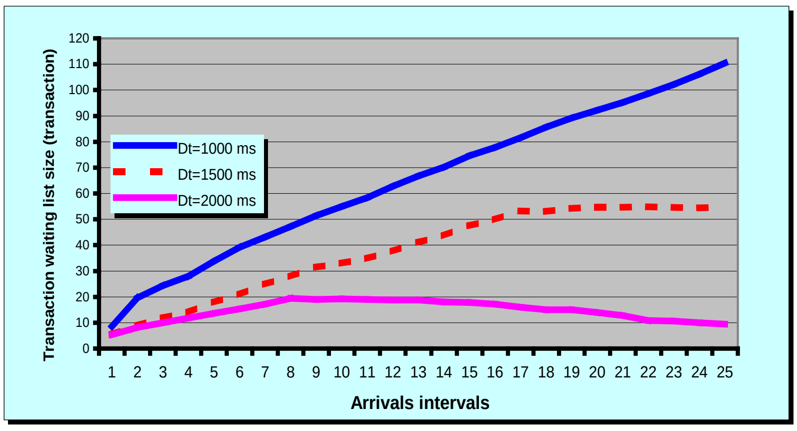

Fig 6.5. *Transaction waiting list variation in function of the arrival rate.*

The above graph shows that for any interval $D_t \geq 1500$ ms milliseconds, the transactions list size remains almost constant starting for a certain moment of the working of the transactions manager.

On the other hand, for any $D_t < 1500$ ms value, a list size increase can be observed. The list would infinitely rise if the transactions manager working in a quasi–permanent mode. In such a situation, the transactions would Riske to wait a long time before set into operation. Litter's law on the waiting list theory states that if T_R L, λ respectively represent the mean reponse time of any random transaction, the transactions average arrivals rate in the execution module, then Litter's law can be expressed by the following formula :

$$T_R = \frac{L}{\lambda} \qquad \textbf{\textit{(6.1)}}$$

3.1.1.1.1.1.1 Where

$$\lambda = \frac{a\ transactions\ stream\ size}{D_t} = \frac{10}{D_t} \qquad \textbf{\textit{(6.2)}}$$

The condition of transactions list stability means

$$D_t \leq 1500 \qquad \textbf{\textit{(6.3)}}$$

Now, it be inferred from the fomula ***(6.2)***

$$D_t = \frac{10}{\lambda} \qquad \textbf{\textit{(6.4)}}$$

According to *(6.4)*, the inequality *(6.3)* so becomes

$$\frac{10}{\lambda} \geq 1500$$

3.1.1.1.1.1.2 Hence

$$\lambda < \frac{10}{1500} \quad transactions = 6.66 \quad transactions$$

For the transactions waiting list to stabilise, it is necessary and it suffices the arrivals rate λ of the transactions the execution module be strictly inferior to 6.66 transactions persecond in average.

7. **Conclusion**

The concurrent access management protocol based on value dates has the advantages of being interlock free, it causes few rejections and therefor provides better performance compared with other existing protocols.

The implemented transaction manager ensures that any transaction in operation in the system will be satisfactorily carried out at the end of a finite waiting time, and provides extra functionality to minimise the chances of livelock and critical failure the VDAS is liable to. To this end, some functionality based on the concept of priority associated with each transaction in operation was added.

The performance measurements and test carried out showed that with a conflict rate between 8.92 % and 15.24 % the transaction manager behaves in a stable way and the streams processing time does not explode until streams with a size equalling 400 transactions.

Furthermore, studies carried out not only enabled the transactions manager behaviour to be observed but also contributed to fix the values of the different parameters allowing a reasonable response time.

Those different achievements later made it possible to more easily face the concurrent access problem linked to a truly distributed architecture, what is the SDDS multiple clients / SDDS servers.

[This work also made it possible for the researches to get accustomed with the JAVA language programming concepts, ore precisely the network programming and that of the multitask in a NT and TCP/IP windows environment

3.1.1.1.1.2

References

[WLH88] Litwin, W., Tirri, H. Flexible Concurrency Control Using Value Dates IEEE Distributed Processing Techn. Newsletter. Special Issue on Heterogeneous Distributed Databases. Vol. 10, 2, Nov. 1988, 42 - 49.

[WLH89] Litwin, W., Tirri, H. Flexible Concurrency Control using Value Dates. *Integration of Information Systems: Bridging Heterogeneous Databases.* Gupta, A. (ed.). IEEE Press, 1989.

- **[LNS94]** W.Litwin, M-A Schneider ,SDDS RP* :A family of Order-preserving Scalable Distributed Data Structures. 20 th Intl.Conf on very large data Bases (VLDB), 1994

- **[AKF72]** A.Kaufmann , Méthodes et modelés de la recherche opérationnelle (tome1) , DUNOD, Paris, 1972

- **[HFA91]** Henry F.Korth, A.Silberschatz , Système de gestion des bases de Données, McGRAW-HILL, 1991

- **[CHT98]** Craig Hunt, TCP/IP Administration de réseau, O'REILLY, Paris, 1998

- **[EDM96]** P.Niemeyer, Joshua Peck , Java par la pratique, O'REILLY INTERNATIONAL THOMSON, Paris, 1996

- **[GPV90]** G.Gardarin, P.Valduriez, Base de données objets, déductives, reparties, EYROLLES, Paris, 1990

- **[SAR94]** S.Miranda, A.Ruols, Client-Serveur, Concepts moteurs SQL et architectures parallèles, EYROLLES, Paris, 1994

- **[AMR99]** A.Mirecourt, Le développeur Java2, OSMAN EYROLLES MULTIMEDIA, Paris, 1999

- **[JBB90]** J.Beauquier, B.Berard, Système d'exploitation Concepts et Algorithmes, McGRAW-HILL, Paris, 1990

- **[AWD98]** A.W.Diéne, Manipulations parallèles et haute disponibilité de structure de données distribuées et scalables, Rapport de recherche UCAD, Dakar, 1998

- **[YBD98]** Y.Ben Adelkader Ndiaye, Architecture de communication des SDDS RP*n et SDDS RP*c, Rapport de recherche UCAD, Dakar, 1998

- **[NAS95]** N.M.Diop, A.Lo, S.Andrianjafy, Adjonction d'un module de contrôle de concurrences par la méthode de verrouillage à deux phases (2 Phases Lock) à un prototype existant, Projet de recherche ESP, Dakar, avril 1995