Publication avant LAFMC

Titre	Deadlock and livelock free Concurrency Control by Value Dates for Scalable Distributed Data Structures
Auteurs	Mouhamed Tidiane SECK, **Samba Ndiaye**, Ibrahima E. Kane, Witold Litwin,
Référence	Proceedings of the Fifth Workshop on Distributed Data and Structures, June 2002 (WDAS 2002)
Editeur	WDAS
Pages	1 - 6
Année	2002
DOI	
URL	https://www.researchgate.net/publication/221193092_Deadlock_and_Livelock_Free_Concurrency_Control_by_Value_Dates_for_Scalable_Distributed_Data_Structures
Index	
ISBN	
Encadreur	Non
Extrait d'une thèse	Non

Deadlock and livelock free Concurrency Control by Value Dates for Scalable Distributed Data Structures

Mouhamed Tidiane SECK[1], Samba Ndiaye[2], Ibrahima E. Kane [3] , Witold Litwin[4],

Extended Abstract :

1. Introduction

A Scalable Distributed Data Structure (SDDS) stores application data in a file transparently distributed over the nodes of a multi-computer, [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 invoke 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 [LNS93] [LNS94]. 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 multi-computer*, i.e., a high-speed network of popular computers [AWD01]. 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.

An SDDS 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

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

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 [RK95] [WLH89]. 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 (2PL) 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 given maximal *p* value ***P***. It also increases adequately with the ratio of *T* completion (current execution time / expected execution time until V). Let again T1 and T2 conflict with current priorities p1 and p2 both smaller than ***P***. The new conflict resolution is as follows:

```
(2)    If V2 > V1 then T2 waits else
       If p1 = p2 then abort and restart T1 else
       Choose for abort and restart T with smaller p.
```

A transaction whose priority reaches ***P*** since it reached ***V*** restarts with the same priority and longest possible value date. All transactions with priority ***P*** are furthermore totally serialized on first-in first-out basis. No livelock can occur in this way.

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, e.g., $\varepsilon = 0.1$ in what follows. The factor ε is a provision for some estimation error and possible waiting time during the execution. Then, the manager first estimates the length L of the transaction to (re)start for *m*-th time as :

$$L = n_R * t_R + n_W * t_W + 2^m \varepsilon * (n_R * t_R + n_W * t_W)$$

The value date V is then computed as $V = L + D$, where D is the current time (date-time) when first operation of *T* is launched by the manager for the actual execution. The operation use the usual services of SDDS-2000 .

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 series we called *streams* of concurrent transactions. The number of transactions per stream was a parameter we have varied. The concurrency resulted from the multithreaded launches of operations . The transactions conflicted randomly on predefined set of up to a few thousand keys to insert into the RP*$_N$ file. For the experiments, 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 bandwidth of each machine had to be shared among the servers, slowing accordingly the response times 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. The characteristic of main interest was the efficiency of the conflict resolution. Especially, the ratio of aborts and of multiple restarts. Since our goal was the concurrency, our 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.

3.1- Test environment

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.

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.

3.2- Study of the transaction manager behaviour in a concurrent environment.

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. We want to make sure that the VDAS algorithm which is the basis of our concurrency control, plays the main role. The priority based module hushes the execution of the small number of potential livelock transactions and guarantees the complete processing of any transaction in a limited time.

The experimental procedure consisted in starting series of updating transactions stream 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 table below shows the duration of the complete execution of the transaction stream (Total execution time in milliseconds), the number of aborted transactions due to the VDAS algorithm (Nb of VDAS aborts), the number of aborted transactions due to the priority based module (Nb of Priority aborts), the number of waiting transactions (Nb of Waiting transactions), and the number conflicting transactions (Nb of Conflicts).

Thus, *Nb of Conflicts = Nb of VDAS aborts + Nb of Priority aborts + Nb of Waiting transactions*

	10	50	100	200	300	400
Total execution Time (ms)	1261	7811	15662	32602	52438	76425
Nb of VDAS aborts	4	40	76	159	254	350
Nb of Priority aborts	0	2	4	0	4	6
Nb of Waiting transactions	5	69	163	335	484	596
Nb of Conflicts	9	111	243	494	742	952

Tab1. Measurements of the number of conflicts and execution time of various set of de transactions

We can compute the different rates using the previous values in *Tab1.* :

Table 2 shows the rate of aborted transactions due to the VDAS algorithm (Rate of VDAS aborts), the rate of aborted transactions due to the priority based module (Rate of Priority aborts), the rate of waiting transactions (Rate of Waiting transactions), and the mean execution time per transaction (Execution time per transaction in milliseconds).

	10	50	100	200	300	400	Mean values
Rate of VDAS aborts	44,4%	36,0%	31,3%	32,2%	34,2%	36,8%	35,8%
Rate of Priority aborts	0,0%	1,8%	1,6%	0,0%	0,5%	0,6%	0,8%
Rate of Waits	55,6%	62,2%	67,1%	67,8%	65,2%	62,6%	63,4%
Execution time per transaction (ms)	126,1	156,2	156,6	163,0	174,8	191,1	161,3

Tab2. Measurements of conflict rates and execution time per transaction.

The latter values can be visualised through this graphical representation. (Fig1)

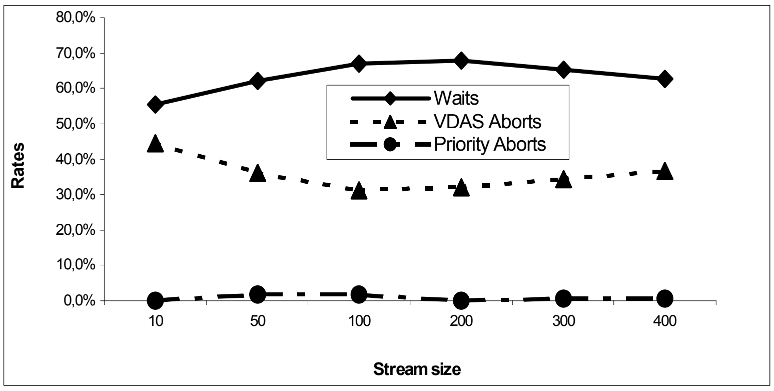

Fig1. Variation of the conflict's rates according to the stream transaction size.

More than 99% of aborts are due to VDAS algorithm and the expiration of value dates during the transaction wait status.

The priority part of the algorithm has a marginal effect in the rate of aborts But it is necessary to avoid livelock. This experimental finding is totally conform to our hypothesises

In this study P=6 and the priorities based extra module is triggered when p>3.
The maximum value of the priority is 6. Beyond this value, the transaction is forced into the sequential execution list. If p is below 3 then complementary module is inhibited.

The influence of the parameter p is dealt with in the complete version of the paper. Our findings show that the rate of VDAS aborts and that of priority aborts are mutually counterbalanced under the variation of p. Nevertheless this process does not affect our first conclusions.

To validate the priority based VDAS algorithm, we measure the number of restarts with a growing size of transaction streams. When a transaction aborts we push it in a list in order to restart it later. The execution of the transaction stream is finished when all the transactions has been completed.

The results obtained are resumed in the table bellow (Tab3.) :

		1 time	2 times	3 times	4 times	5 times	6 times
Stream size	**10**	4	0	0	0	0	0
	50	19	7	3	0	0	0
	100	41	12	5	0	0	0
	200	106	25	1	0	0	0
	300	194	22	4	2	0	0
	400	262	24	10	4	0	0

Tab3. variation of the number of restarted transactions according to the frequency of restart for various stream size .

This histogram represents the values in the precedent table (*Tab3.*) :

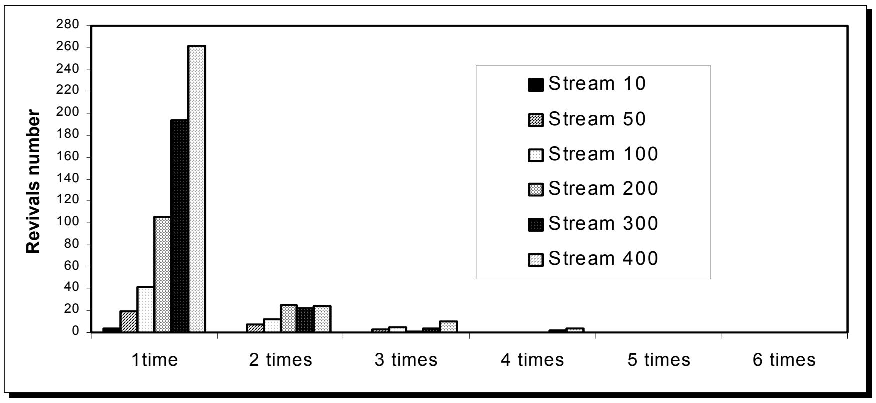

Fig2. variation of the number of restarted transactions according to the frequency of restart for various stream size .

Our findings show the effectiveness of the priority based module in the processing of the potential livelock transactions. No one transaction has been restarted more than four times. Hence none of them is forced in the sequential execution list.

4. Conclusion

The concurrent access management protocol based on value dates has the advantages of being interlock free, it causes few rejections and therefore 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 *(full version of the paper)*.

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.

REFERENCES

[AWD01] Ali W. Diene Contribution à la gestion de Srtuctures de Données Distribuées et Scalables, thèse de doctorat Université Paris 9 Dauphine 2001

[LNS93] W.Litwin, M-A Neimat, D. Schneider LH*: A Scalable Distributed Data Structure (Nov. 1993). Submitted for Journal publ.

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

[RK95] R. Karlsen, Flexible Transaction Management in Federated Database Systems, Ph.D. theses, Department of Computer Science, University of Tromsø, 1995

[WLH88] W. Litwin and H. Tirri. Flexible Concurrency Control Using Value Dates. Technical Report 845, INRIA, May 1988.

[WLH89] Litwin, W., Tirri, H.: Flexible Concurrency Control using Value Date. In Gupta, A. (Ed.): Integration of Information Systems: Bridging Heterogenous Databases. IEEE Press, 1989.