An actor like data model for a parallel DBMS

W.K HIDOUCI - D.E ZEGOUR
Institut National d'Informatique, (I.N.I)
BP 68M Oued-Smar, Algiers (Algeria)
{hidouci , d_zegour}@ini.dz

Abstract:
We present in this paper the new concept of "*actor databases*" (*DB-Act*) being studied at the INI institute (Act21 project) where we are developing a parallel main memory database system based on an actor like data model. To achieve data distribution we use a Scalable Distributed Data Structures (SDDS) called "*distributed Compact Trie Hashing*" (*CTH**) currently in development at the same project.
Transaction management and recovery techniques are adapted for the combination of actors and SDDS in Act21. An adaptation of the nested transaction model to the actor paradigm is presented, as well as a recovery techniques using a fuzzy checkpointing tailored to the SDDS needs.

***Key-Words:* -** Actor paradigm; Scalable Distributed Data Structures; Parallel DBMS; Recovery

1. Introduction

Main memory databases (MMDB) are an attractive solution for database applications which require very high throughput and fast response time (telephone switching, real time applications, …) [9], because data reside in main memory and secondary memory accesses are only needed for recovery purposes (namely : logging and checkpointing) [14, 15, 17]. Another way to improve performances in database systems is via parallelism and distributed computing [7, 8].

Scalable Distributed Data Structures (SDDS) provide a scalable data partitioning in distributed RAM and can be used as a storage sub-system for a parallel and/or distributed MMDB. This is a class of distributed data structures developed initially at Paris-Dauphine University (ceria[1] laboratory) and devoted to handle very large data files in a distributed environment [20]. Each RAM file is composed by a set of SDDS-servers that store records in main memory at each node. Application programs that access these files are called SDDS-clients. SDDS methods can be considered as distributed file structures offering very high performances and scalability. Some of them are order preserving [21, 26] (like range partitioning, each server is related to an interval of key values). Some others can be extended to provide high availability [19], i.e. despite failure of some servers, the file is still totally available (the maximum number of faulty servers being a parameter).

Actor programming paradigm was introduced and popularised by the works of researchers mainly in the area of distributed AI [1, 11, 24]. Actor languages are often used in the development of distributed open systems where each component is described by an autonomous dynamic object (called 'Actor') that encapsulate a part of the global knowledge. To achieve a common goal, actors must collaborate by means of messages passing. The system can be extended or modified dynamically by creating new actors and inserting them into the running system without global reorganisation.

Our goal in "Act21" project is to build a parallel DBMS that use SDDS to achieve data partitioning and "actors" to handle distributed processing. We focus in this paper on the actor data model and on transaction management as well as checkpointing and recovery techniques.

As far as we are aware of, it is the first time that SDDS are used to build a parallel DBMS. There were two previous works that concern the coupling of SDDS techniques with existing DBMS [18, 22]. Their main purpose is to add some scalability to the database system by maintaining a partitioning scheme dynamically adaptable to the size of the database.

There are two major contributions of the present work:

- We show that actor paradigm is well suited for parallel database applications. An actor like data model is presented, having the same functionalities as the object model (encapsulation, inheritance, ...) and providing some facilities for parallel processing.
- We present an implementation of nested transactions and recovery techniques adapted to data management with SDDS methods.

The remainder of this paper is organised as follows : In section 2 we present our system model. The next section presents our actor like data model. SDDS used as main memory storage sub system are presented in section 4. Transaction management and recovery for our actor model are described in section 5. Section 6 provides some justifications to our design approach. Section 7 gives our first experimental results and some implementation issues. Section 8 presents some related works and section 9 concludes the paper.

2. System model

The database is distributed over a set of networked computers and is composed by several "actors" which communicate together by message passing and manipulate data stored in distributed RAM-files using Scalable Distributed Data Structures (SDDS).

Each node (site) maintains a set of actors and of SDDS-servers (see Figure 1). Some actors (see later T-Act) communicate with SDDS-servers to store and manipulate data. These are considered as SDDS-clients.
An SDDS-server manage a bucket consisting of a fixed number of pages (I/O transfer unit). Pages contains records of the form : <key, attribute_value> .

[1] http://ceria.dauphine.fr/

We assume that the size of main memory is sufficient for all the SDDS-servers and all actors for a particular site. The entire database is then kept in distributed main memories. The set of disks attached to each site are used only to perform checkpoint and logging activities.

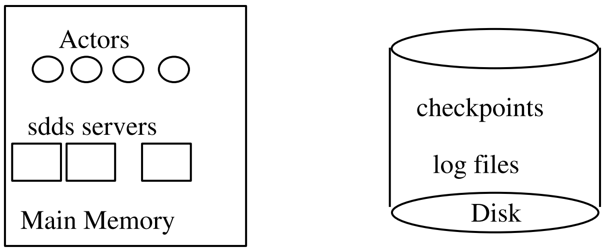

Figure 1: Architecture overview

3. The Act21 data model

Act21 DBMS is based on a actor like data model (more details are to be found in [12]). It is mainly based on the notion of "databases actor" introduced below.

We consider two types of objects:

- The data (passive objects) represent the information stored in the database. Their structure is defined by their type. Every data is identified by a unique and transparent Object IDentifier (OID) but contrary to the object model, doesn't have a behavior. In our case, the dynamic behavior is associated to the type itself that is considered as being an active object (see T-Act below). The same approach has been adopted in the model PRACTIC based on the object model [2].
- The database actors (DB-Act) that are autonomous dynamic objects (tasks) defined by a set of attributes (*data*) and a set of applicable methods (*script*). Actor's identity is unique (OID), besides, a symbolic name can be assigned to any actor. The communication between actors is generally asynchronous (the sender doesn't wait for a response from the receiver).

Data part and script part define the *behavior* of an actor. It is equivalent to the class definition in the object model [5,6].

There are three types of DB-Act:

- Type actors (T-Act): These actors represent user data types. Their role is to maintain and manage the stored data and to answer queries from other actors in the system. (see Figure 2 for an example)
- Collection actors (C-Act): These actors are containers. They allow to store a collection of data and can be used to represent some multivalued fields or to store temporary results of queries.
- Request actors (R-Act): These are programming actors. They don't have predefined role as the two first, their behavior must be specified entirely by the user or by an application program.

Every actor is characterised by: an OID, a type (T-Act, C-Act or R-Act), a behavior (optional in the case of a C-Act) and a name (obligatory in the case of a T-Act). The T-Acts and the C-Acts are endowed with predefined methods. These operations define the default behavior of this type of actors (instances and schemas management, generic collection manipulation methods, ...).

For T-Acts (actors representing types), every field described in the Data part is considered as an attribute of the new type and methods in the Script part, are operations allowed to manage the type instances.
The behavior (Data and Script) is described with a programming language (PACT) syntactically close to C++ and permits to specify actor's fields (Data) and the applicable methods (Script) in addition to those already predefined. A general form for a behavior declaration is :

```
Def_Behavior BEHAVIOR_NAME {
Data :
   // fields declaration ...
Script :
  // methods declarations ...
} ;
```

One of the main features of the actor programming paradigm is the use of *asynchronous* messages, i.e. the sender sends the message and continues its execution concurrently, without waiting for a particular result from the receiver. On the other hand one can specify in the asynchronous message, the actor (called a *continuation*) having to receive a possible answer. This makes possible a serial execution of tasks, achieved by successive continuation actors [1].

Synchronous messages suspend the sender execution until the receiver completes its task and returns a result to the sender.

Predefined types are i*nt, float, string, bool* and *OID.*
getValue(…), setValue(…) and *select(…)* are some T-Act predefined methods.

```
Def_Behavior TypeEmp {
Data :
        int ssn ;
        float  salary ;
        string department ;
        OID  manager =NULL;
Script :
        managerName( OID  e ) :  C  {
           string s ;
           OID x = getValue('manager',e);          // retrieve value of the field 'manager' in instance e
           if (x != NULL) {
                 s = getValue( 'last_name' , x );   // retrieve value of the field 'last_name' in instance x
                 send string(s) to C ;              // send the name to actor C
           }
        }
        DepartmentEmp( string dep ) :  C  {
          select('department = ' + dep) : C
        }
} ;

Def_Behaviour TypePers {
Data :
        string last_name, first_name, address ;
        int age ;
 } ;
```

Figure 2 : Examples of T-Act behaviors

In Figure 2, an employee's behavior (TypeEmp) has been described by the fields: ssn, salary, department and manager. This last being of OID type initialised to NULL. The script defines two methods (managerName and DepartmentEmp) : one returning the manager name of a given employee (e) and the other returns the set of the employees working in a given department (dep). One can make the difference with class methods (in the object model) where methods manipulate only one instance (object) whereas in a T-Act, a method manipulates the whole instance of the type. A second behavior, in the same figure, defines the type Person (TypePers) as being an aggregation of some conventional fields (last name, first name, address, age) without methods.

For type actors (T-Act), fields defined in the Data part of the behavior, are only accessible by predefined methods (*setValue* and *getValue*) in the Script part, achieving thus the principle of encapsulation.

Two asynchronous message types exists:

- *send message [: C] to A* : In this case the message is sent to actor A and the execution continues. If a continuation is specified (actor C), the result of the message will be sent to it by actor A, in an another asynchronous message.
- *broadcast message [: C] to G* : The message is diffused to the actors belonging to the specified group (group G). The actors can join and leave groups dynamically. As previously a continuation actor (C) can be specified in the message.

In both cases, the message is formed by the name of a method with its possible parameters.
If the message is in the form *var = send message to A* it becomes then *synchronous*, i.e. the execution is suspended until the result is returned to the variable '*var*' (that plays the role of a blocking continuation).

Inside of an actor's script, one can remove the expression '*send to*' if the message is destined to the actor itself. For instance in method *managerName* of Figure 2, the instruction *getValue('last_name',x)* sends the message to the actor it self to get the value of the attribute *last_name* associated with OID instance *x*.

If the message received by an actor doesn't exist in its script, the message is redirected to the delegated actor (or *proxy*). By default the delegate of a new actor is the predefined actor '*Error*', but one can change the delegate at any time by the primitive '*ch-deleg(a1,a2)*' or when creating the actor's by the primitive '*new_act (…,OID-proxy)*'. This mechanism permits the definition of dynamic inheritance between actors.

3.1 Type actors (T-Act):

Some primitive types exist (I*nt, Float, String*,…). One can build other types by creating T-Acts. The new T-Acts correspond to the new user defined types.
Every T-Act is described by its behavior that specify the new type attributes and the instances management methods. The new methods in the behavior are added to the predefined ones.
Among the predefined methods of the T-Acts, we can cite :

- '*add_inst(v1,v2,…) : c*' Insert a new instance and returns (to c) its OID.
- '*del_inst(OID)*' Delete an instance (OID).
- '*setValue(OID,field,val)*' Updates the value of the specified field for the specified instance.
- '*getValue(OID,field):c*' Returns to c, the value of the specified field for the specified instance.
- '*select(condition):c*' Returns to *c,* a set of instances of this T-Act verifying the specified condition. In that case the set of instances transmitted to c is terminated by the message '*end(OID of the T-Act, nb)*' where nb designates the number of elements transmitted.

For example we can define a T-Act named "Employee", while using the behavior of Figure 2 by the following primitive :

Emp = New_Act('TypeEmp', 'Employee', T-Act);

The variable *Emp* represents then the OID of the new T-Act managing the type "Employee".

3.2 Collection actors (C-Act)

A collection is a container actor (C-Act) managing a set of instances (passives objects). The predefined methods of C-Acts are those generally used in the generic collections (Cardinality, Exist, Insert, Remove, …). *Groups* are particular C-Acts containing other database actors (T-Act, C-Act or R-Act).
Among the predefined methods of C-Act :

- '*card( ):c*' return to *c* the cardinality of the collection.
- '*<type>(val)*' where *<type>* in {*int, float, char, bool, string, OID*}. Its purpose is to insert '*val*' into the collection.
- '*pop( ):c*' remove an element from the collection and send it to *c*.

Figure 3 shows an example of behavior that can be used to create a collection actor that print a field value (FN) of instances sent by a T-Act (TN) in response to a select message.

```
Def_Behavior  C1  {
Data:
    string TN ;              //  a type name
    string FN;               // a field name

Script :
    init( string t,f) :c  {    // a method that initialise the fields TN and FN
        TN = t ;
        FN = f ;
        reset( ) ;             // Reset the content of the collection (a C-Act predefined method)
        send int(0) to c ;     // send 0 to the continuation
    }

    end( OID e, int nb ) { // method redefinition. The default end() method do nothing.
        print(nb) ;
    }
    print(int nb)  {            // a method that prints the content of a collection
        OID x, p ;
        string n ;
        p = Get-OID(TN) ;                    // a primitive to get the OID of T-Act named TN
        while ( card() > 0 )  {              // For each item in the collection
            x = pop() ;
            n = send getValue(x,FN) to p ;   // get back the  value of the field  FN
            send string(n) to cout ;         // and print it.
        }
    }
};
```

Figure 3 : Example of a collection behavior

Using the behavior of Figure 3, a collection actor is allocated by the primitive :

E=New-Act(C1,C-Act) ;

Then, C-Act *E* can be used as a selection result addressed to a given T-Act (TN) to display the values of a given field (FN). When the message e*nd(OID, nb)* is sent to *E*, the method '*print(nb)*' is activated to display the values of the FN field of the instances that have been received.

3.3 Request actors (R-Act):

A request is an actor (R-Act) defined outside the DBMS by the user (or by an application program) or inside by the query sub-system of the DBMS. Its role is generally to communicate with the other actors of the system to find the information requested in the database.

R-Acts have no predefined methods since they are used for programming purposes. Their role must be entirely coded in a behavior by an application programmer. Figure 4 shows the behavior of an R-Act that uses a C-Act to select and to display the names of people aged more than 40.

```
Def_Behavior  req  {
Data:
    OID col;

Script :
    Run()  {
      int x;
      OID p;
      col = New-Act( C1 , C-Act ) ;                    // Create a collection (see Figure 3)
      x = send init("Person", "last_name") to col ;  // and initialise it
      p = Get-OID( "Person" ) ;                        // Get the OID of the T-Act named 'Person'
      send select("age > 40") : col  to p ;            //  and sends a selection message
                                                       // The result will be redirected to the C-Act col
    }
};
```

Figure 4 : R-Act example

One will be able to use this behavior to build an R-Act and to call its *Run* method

R = New_Act(req, R-Act) ;
Send Run( ) to R ;

When *R* receive the '*Run( )*' message, it creates a C-Act (*col*) and initialises it by the message '*init("Person", "last_name")*'. The form of the message '*x=send…*' (synchronous) is only used to make the actor wait until *col* is completely initialised. After that, we get the OID (*p*) of the T-Act "*Person*" and send a select message where we specify *col* as a continuation. Therefore the OIDs of persons that match the condition "*age > 40*" will be sent to *col.* As soon as *col* receive the '*end(…)*' message from *p*, indicating the end of the select, it prints the names of the instances to the console (*cout*).

3.4 Methods overloading

When defining a delegation relationship between two actors A1 and A2 so that A2 is the proxy of A1, this last can overload one or several methods of A2 to adapt them to its own behavior, it's the same concept as the method overloading in the object model.
The predefined methods for the different types of actors can also be overloaded while specifying a new behavior. This possibility is important because it permits to adapt the system to a particular use or to define constraints on the field values and triggers to control them.

Inside the body of the overloaded method, one has the possibility again to use the original predefined method. All occurrences of the predefined method in the body of the overloaded method is implicitly linked to the predefined ones.
For instance, if we want to make some checks (constraints) on data insert or update, we have to redefine in a T-Act the method 'setValue' when specifying its behavior or by dynamically adding the overloaded method later :

```
Def_Behavior … {
Data:
    …
Script:
   …
    setValue(…) {// the redefined method
           // test and/or make something …
           …
           setValue( …)        // call to the original method
    }
};
```

4. The storage sub system

Data manipulated by T-Acts (the whole instances of the database) are stored in distributed RAM-files. Each RAM file is a Scalable Distributed Data Structure based on CTH* (distributed Compact Trie Hashing). The reader can consult [25] for more details on the method.

SDDS (and thus a distributed RAM-file) is a collection of "servers" disseminated over the network nodes, each server maintains a bucket (in RAM) to store records. While the file is accessed (searches, insertions and deletions) by "clients" programs, the number of servers grows and shrinks adequately to preserve good performances.

Formally, a file structure that meets the following constraints is called an SDDS [20] :

- ➢ A file expands to new servers gracefully, and only when servers already used are efficiently loaded.
- ➢ There is no master site that record address computations must go through.
- ➢ The file access and maintenance primitives (search, insertion, split, ...) never require atomic updates to multiple clients.

Generally, in the SDDS methods, the address computation function (that translate record keys to server addresses) is globally unknown. Thus clients can make addressing errors while accessing the file, however, servers can collaborate to redirect the client operation to the correct server. The client can than adjust its partial addressing function accordingly.

In CTH* method, the file is ordered and partitioned among several servers. Each client maintains a small binary radix tree (called 'Trie' : a kind of binary search tree indexing the RAM-file, where each node contains only one digit of the key) as an addressing function. On each server there are:

- ➢ A radix tree (that keeps track of all the splits in this server)
- ➢ A bucket containing records;
- ➢ An interval]key_min , key_max] indicating respectively the lowest and the highest keys stored in this server.

The expansion of the file is done through collisions that occur when inserting a new record in a full server. At each collision a new server is allocated in the system. Records in the old bucket are redistributed between the two servers (split operation). Intervals and trees are updated accordingly. The client initiating the insertion is also updated.

Any new client in the system starts with an empty tree that maps all keys to the first server allocated to the file. If a client contacts a wrong server, it updates its image using the server tree. Successive file accesses make the client image converge gradually to the real global tree.

5. Transaction management

Transaction management and recovery concern specifically type actors (T-Act) and SDDS-servers, because they represent the distributed database. We recall that the whole T-Act instances are managed by the SDDS-servers. Thus we have to adapt the conventional transaction architecture [3] where each site maintain one Transaction Manager (TM), one Concurrency Controller (CC) and one Data Manager (DM) to an architecture where each site maintains several TM, CC and DM.

Each T-Act method execution is implicitly considered as a transaction. Concurrent execution of actors imply that access to T-Act instances (data) must be scheduled to produce strict serializable execution. Concurrency controllers (CC) and data managers (DM) are embedded in SDDS-servers, while transaction managers (TM) are part of actor predefined behaviors (thus in SDDS-clients). There is also a Recovery Manager (RM) (one per site) that collaborate with all the DM located in the same site to perform checkpointing during normal operations and recovering after a crash.

When a T-Act executes a method, the TM takes charge of data manipulation operations (getValue / setValue), computes server addresses using its addressing function and then sends the associated (read/write) operations to the CC of the appropriate SDDS-server. The TM also, keeps track of distributed transaction execution using 2PC (two phases commitment protocol).

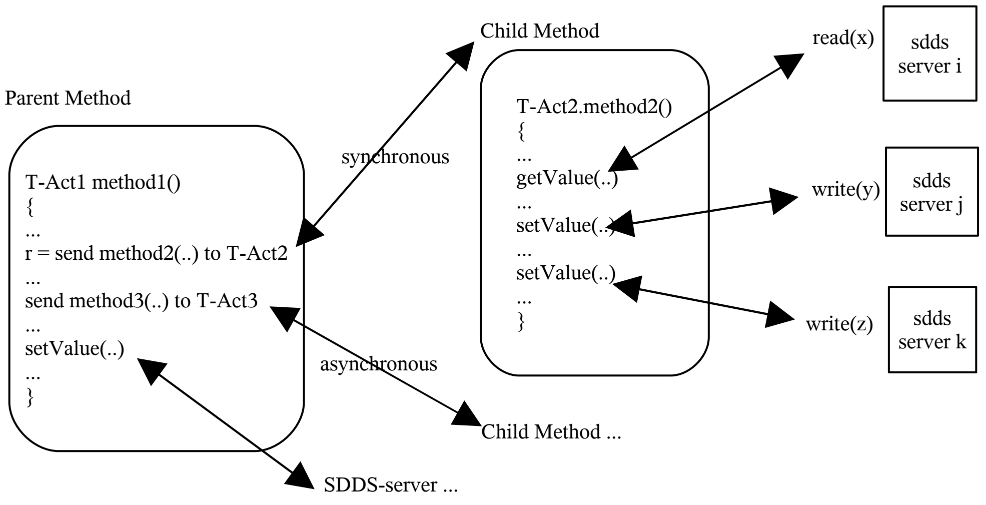

Figure 5 : Transaction management in Act21

Nested transactions occurs, in our architecture, when a (parent) method in T-Act x sends synchronous message to T_Act y (child method). The parent method's locks are “offered” to the child method. This is the “downward inheritance of locks” presented in [10]. When the child method terminates, the new locks that have been acquired are transmitted back to the parent (“upward inheritance”). For asynchronous message however, the parent and child methods are completely independent. Locks are not transmitted between parent and child methods in that case.

For instance, the execution of 'method1' in Figure 5 is considered as a parent transaction. 'method2' of T-Act2 and 'method3' of T-Act3 are two sub-transactions. Low level operations (read / write) are executed by the SDDS-servers (CC and DM modules).

The commitment of a transaction depends on the commitment of all of its children (both synchronous and asynchronous ones), if one child aborts, the parent must roll-back. 2PC can be easily extended to handle this nested approach.

Each SDDS-server incorporates a scheduler (CC) and a data manager (DM). CC implements strict 2PL, i.e. two phase locking protocol where transactions hold their locks until they terminate (commit or abort). Locks can be shared (S) or exclusive (X) and are managed at page level. The DM executes the (read/write) operations on pages located in its bucket. No I/O are needed to perform this task (data is memory resident), however the logging and checkpointing activities may require the DM to access the local disks (e.g. for transaction commitment).

Physical Logging

SDDS-servers maintain active transaction tables (ATT) (one per server) that keep track of transaction execution (see Figure 6).

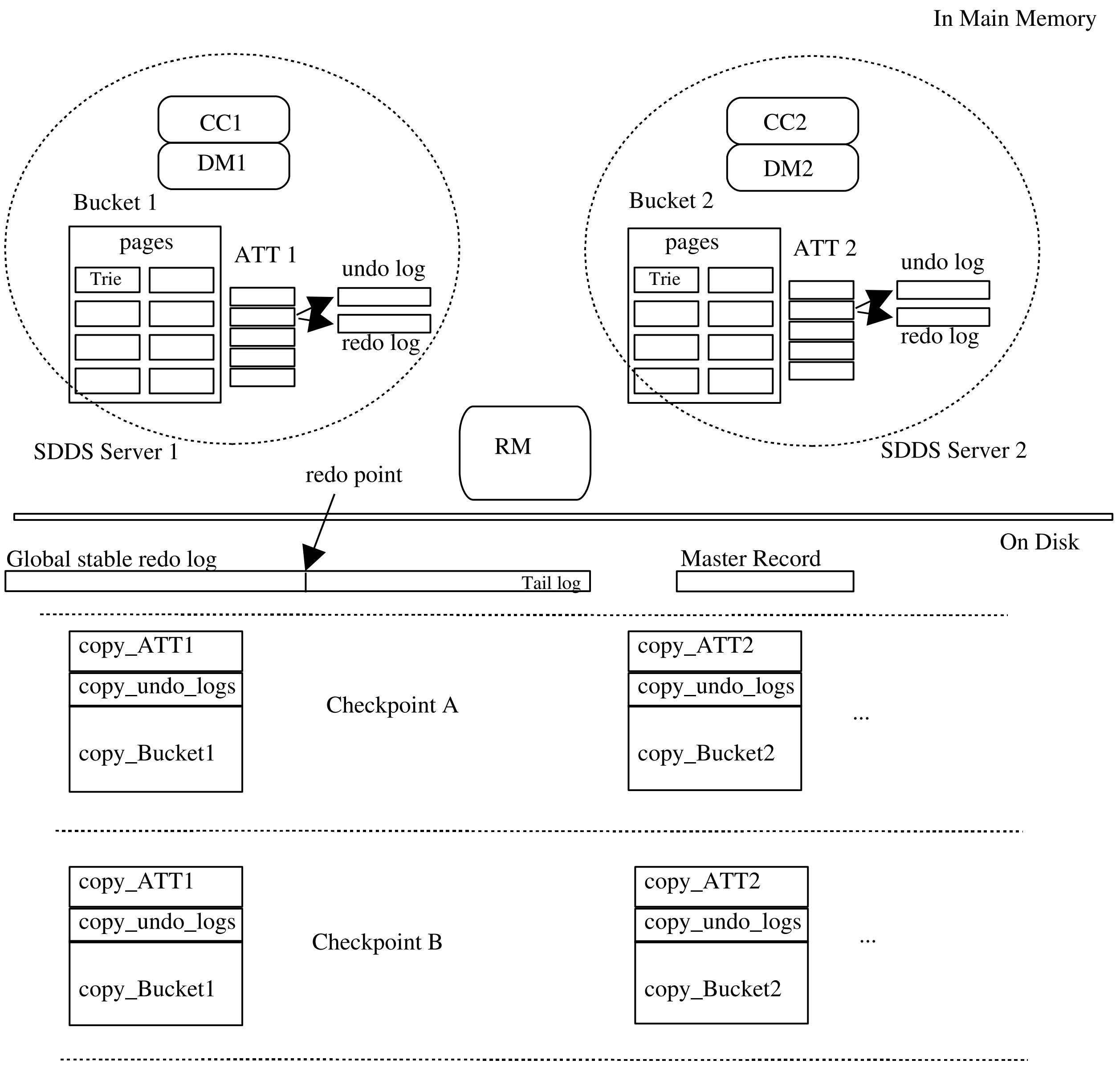

Figure 6 : Node storage sub-system

Each ATT entry contains both the undo and redo logs for one transaction. The undo log is used to roll-back the effects of a transaction if it is aborted, whereas the redo log is flushed to the global stable log (on disk) when the transaction commits.

All update operations on a page x generate physical undo record : <x, pos, len, old_value>
and a physical redo record : <x, pos, len, new_value>
where pos and len indicate respectively the position and the offset, in the page, of the updated bytes.
The radix trees (the servers addressing functions) are stored in the first page of each bucket. Thus tree updates are also physically logged and can then be recovered.

When a transaction Ti commits, a record <Ti, Nb_rec> is added to the head of its redo log and then the whole redo log is flushed to the global stable log on disk. Nb_rec indicates the number of log record pertaining to transaction Ti.
The stable log contain only the effects of committed transactions in serial order equivalent to the concurrent (but serializable) execution of these transactions in the MMDB:

```
...   <T1,3>, <x,...>, <y,...>, <z,...>, <T2,2>, <x,...>, <y,...>, <T3,4>, <x,...>, <y,...>, <z,...>, <t,...>   ...
      |______ T1's redo log ______|   |__ T2's redo log ____|  |__ T3's redo log ________________|

                                  the global stable log on disk
```

When a transaction terminates (commit/abort), its entry (and its log records) is discarded from the active transaction table ATT.

Fuzzy Checkpointing:

To recover from failure (system crash), where the content of main memory is lost, the global stable redo log can be replayed from the beginning or (if possible) from a "previous consistent copy" of the database to bring up the latest consistent state before the crash. There are two major drawbacks with this "naive" approach :

- Making a consistent copy of the database is a very time consuming task, because transaction processing have to be stopped during the flush of the buckets content.
- The huge number of records in the stable log makes the redoing procedure very inefficient, when recovering from a crash.

Instead of making a consistent copy, we can "emulate" it by doing a snapshot of the buckets content while active transactions are processed concurrently (the backup copy obtained is then not consistent, since uncommitted updates could be copied out). Then a copy of ATT (active transaction tables) with the undo logs is made. Indeed undoing the effects of uncommitted transactions from the inconsistent database copy makes it consistent. This is a "fuzzy" checkpoint, since the dump procedure do not require the system to be quiescent, i.e. a page locked (even exclusively) by an active transaction, can be dumped during the checkpoint.
In Act21 we have adapted for the SDDS, a variant of a fuzzy checkpoint (called ping pong checkpoint [15]) that keeps two copies of the database on disk. Each one contains an inconsistent dump and the undo logs of active transactions during the dump.

For each SDDS-server we maintain on disk, two copies (A and B) of the bucket, the ATT and the related undo-logs. A master record is also maintained and is composed by : an indicator of the current checkpoint (A or B), a pointer to the redo point in the stable log (the start point for scanning the stable log when recovering) and a "Directory" that keeps informations necessary for rebuilding all the SDDS-servers and T-Act (SDDS-clients) hosted in the site, when recovering.
Recall that buckets are composed of a set of pages (that contain data records). A page is "clean" when no updates occur in since the previous checkpoint. A page is "dirty" when one or more transactions update its content since the previous checkpoint.
The page state (clean/dirty) is indicated by two state bits : 00 for clean and 01 or 10 for dirty.
When a checkpoint occurs, all dirty pages (state bits = 01 or 10) are flushed to one copy of the disk database and their state bits are incremented modulo 3, i.e. those having state bits = 01 remain dirty (10) and those having state bits = 10 become clean (00).
When the next checkpoint occurs, the old dirty pages (10) and new ones (01) are flushed to the other copy of the disk database. Thus each dirty page is flushed twice, once to each copy of the database, on two consecutive checkpoints. If the system crash before completing the current checkpoint, the other copy can be used to recover the database to the latest consistent state.

The checkpoint procedure below, runs periodically without interfering with normal transaction processing :

1. Let x = (A or B), such that x is different from the current checkpoint (stored in the master record)
2. Note the end of the stable log in a variable (redo point)
3. For each bucket in the node, do:
    - write dirty pages to the bucket copy x (on disk)
    - write ATT and the undo-logs to their copy x
4. replace the master record by the new one : < x , redo-point, Directory > (in one atomic operation)

To recover from a crash, RM loads the current checkpoint and roll-backs the transactions that were active during this checkpoint. Finally we replay the global redo log beginning from the redo-point (saved in the master record) to reach the latest consistent state before the crash.

The recovery procedure is as follow :

1. read the current checkpoint (A or B) in x, the redo point and the directory from the master record
2. rebuild the SDDS-servers
3. For each SDDS-server, do:
    - load its ATT and undo-logs from copy x on disk
    - load the bucket pages and reset their state bits (00)
    - roll-back the active transactions from the loaded ATT, using the undo logs
4. Replay the global redo log and update the ATT, undo-logs, and dirty bits accordingly.

Before recovering, the SDDS-servers must restarted with an empty radix tree, when the recover procedure terminates, the latest state of each radix tree will be retrieved from the checkpointed pages (recall that the first page of each bucket is reserved for holding the server "Trie").

SDDS-clients (T-Act) can be rebuilt just after the termination of the recovery procedure, however their radix trees remain empty, until they access again the database. The SDDS methods allow the clients to adjust their addressing function gradually during the file manipulation without performances degradation.
Other actors (R-Act and C-Act) are not concerned by the recovery mechanism, since they are not part of the distributed database (they don't store persistent data).

6. Design justifications

High performances database systems adopt generally the relational data model, because it's simple and well suited to parallelism. However, some database applications need a powerful data model and require high performance too. In our case, we have used the actor programming paradigm as a framework to develop a parallel DBMS offering a semantically rich data model (providing the same functionalities as the object oriented model) and implementing techniques used in high performances database systems. Indeed, actors are autonomous objects used to build open parallel systems. We have just added the persistence property to actors. This were done by incorporating the SDDS techniques in type-actors (T-Act) that manage the entire database.

In high performance database systems, transaction management is among the most significant points to take into account (with storage sub-systems and plans execution optimizations). We have used concurrency controls adapted to main memory resident data. In such a case, where no input/output are needed to process transaction operations, there is very little lock contention as noted in [9], and then, Strict 2PL controllers without fine granularity is among the best choices.

Using SDDS as storage system makes the data distributed scheme self adapting to provide good load balancing when the size of the database change. This avoid many data skew occurrences.

The drawback of memory data residence is its weak fault-tolerance, thus the need of logging and checkpointing updates to stable storage in a way that minimize the interference with normal activities. We have adopted the physical logging with fuzzy checkpoints because we have to flush more frequently, changes to stable storage. In case of system crash, the recovery procedure is then very efficient and can bring the system up more efficiently than traditional checkpoint procedures. The counter part, is of course the relatively short period of successive checkpoints. But with the fuzzy approach, these activities are done in parallel with the normal transaction processing and do not impact really the system performances. More over maintaining two different bucket copies on disk, avoids the management of the undo records in stable storage. The recovery procedure is then more shorter (the traditional undo phase is not necessary).

7. Implementation issues

We have implemented some modules of Act21 prototype in a virtual parallel environment (networked Linux boxes + PVM) :

- An SQL interface that produce an execution plan from an SQL query, consisting of some R-Act and C-Act that cooperate with the existing T-Act to produce the query result.
- An interpreter of PACT language for executing actor's scripts.

- An actor manager that implement the actor's primitives (New_Act, Send, Broadcast, ...) and the transaction management (TM) for T-Act (coordinators in 2PC).
- A storage manager based on CTH* SDDS method. Concurrency controllers, data managers and cohorts (in 2PC) are also implemented in the SDDS servers.
- A recovery manager that uses fuzzy checkpoints is also implemented.

An important module is not yet implemented and concerns the optimization of the execution plan produced by the SQL interface. This issue will be treated in the near future.

We have conduct some preliminary experiments to validate our transaction management scheme. The results are presented below:

We have fixed the size of the bucket to 400 pages of 1KB for each SDDS server.
Our (low-end) parallel virtual machine is composed by 4 PC (Pentium 3 750Mhz – 256MB RAM) connected by an Ethernet switch (10-100 Mb/s). Clients and servers are PVM tasks.

Each client generate 1000 serial transactions, each one is composed by a random number of I/O operations varying from 1 to 10.

example : read page(103) from server(1), write page(82) to server(4), write page(261) to server(3), ...
page and server numbers are also randomly generated .

The tests consist of launching n parallel clients and to observe some parameters like response time, throughput, and the number of aborted transactions for dead-lock prevention (wait-die). When a transaction is aborted, the client wait 10 ms before restarting the transaction.

The response time is computed as the mean of the transaction execution time (from begin transaction to commit). The throughput is the number of successful commitments done in 1 second in the system. It's computed as the sum of all client throughput in the same period.

The next table (Table.1) resume the tests for 2 servers:

# clients	***Resp. Time***	***Throughput***	***Aborts***
10	23 ms	425 tps	1.6 %
20	41 ms	481 tps	3.9 %
30	66 ms	450 tps	7.3 %
40	89 ms	440 tps	10.7 %
50	114 ms	400 tps	15.4 %
60	140 ms	420 tps	20.6 %

Table 1 : Performances with 2 servers

The results observed for 3 servers, in the same conditions, are presented in Table.2

# clients	***Resp. Time***	***Throughput***	***Aborts***
10	18 ms	544 tps	1.1 %
20	34 ms	566 tps	2.6 %
30	54 ms	540 tps	4.3 %
40	73 ms	521 tps	7.2 %
50	91 ms	510 tps	9.8 %
60	114 ms	481 tps	13.1 %

Table 2 : Performances with 3 servers

Finally Table.3 gives the results for 4 servers.

# clients	*Resp. Time*	*Throughput*	*Aborts*
10	18 ms	535 tps	0.5 %
20	29 ms	659 tps	1.9 %
30	44 ms	658 tps	3.1 %
40	59 ms	644 tps	4.7 %
50	76 ms	620 tps	7.4 %
60	93 ms	603 tps	9.6 %

Table 3 : Performances with 4 servers

When the number of aborts due to deadlock prevention is too high (> 4%), the system is in its critical state. The size of buckets (number of pages per bucket), must grow. If it is not possible, the number of server (the size of the parallel virtual machine) must be higher.

In our case (with a low end configuration) the best performances where obtained when the number of parallel clients is about 20, the throughput is then maximal (481, 566 and 659). But if we change the size of buckets, the number of servers or the characteristics of the processors, this number (parallel clients) can be higher.

The response time increase by a factor of 2.34 for 2 servers, 1.92 for 3 servers and only 1.5 for 4 servers, when the number of parallel clients grow from 10 to 60. Figure 7 shows how the response time behave when the number of parallel clients range from 10 to 60 with 2, 3 and 4 servers.

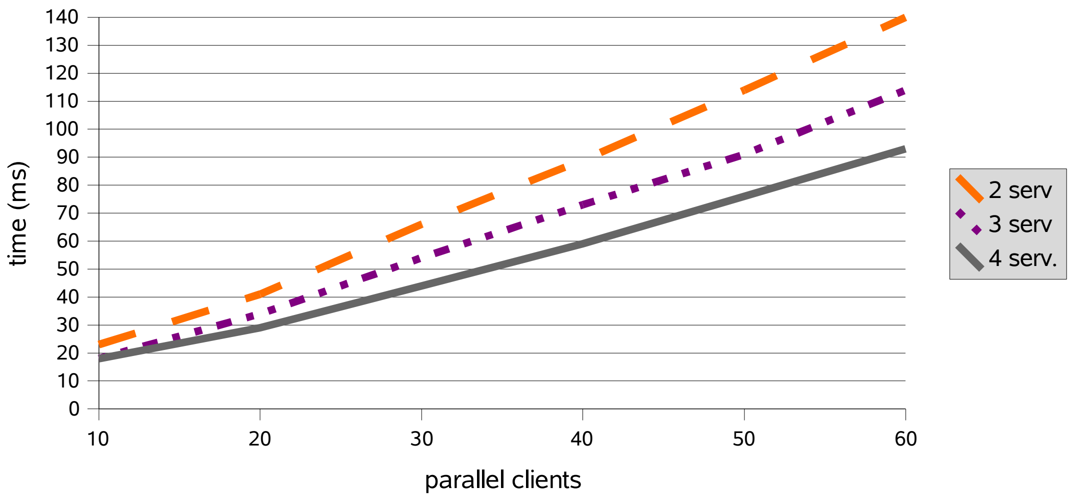

Figure 7 : Response time behaviour

On the other hand, when the abort proportion is not high (the number of parallel clients vary from 10 to 30), the effect of the servers number is very positive. Figure 8 shows that the response time decreases with the number of servers. The amelioration is well noticed when the number of parallel clients is 20 and 30. For 10 parallel clients, the performances stabilise (around 18 ms) which is approximatively the time needed for the log flushes during 2pc. The growth of the servers number do not enhance further the performance.

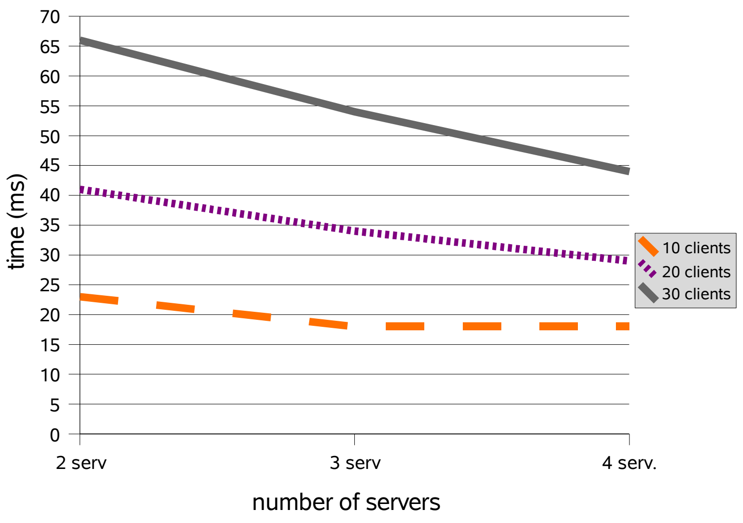

Figure 8 : Performance amelioration

We notice that the speed-up is about 66% to 79% for the response time when the number of servers double (from 2 servers to 4 servers). In the same time the percentage of aborted transaction is decreased by a factor of 1/3 to 1/2. The throughput computed in this experiment if relative to one server only and the global throughput is expected to be higher when the number of servers increase.

8. Related works

A lot of works have been conducted in the area of high performance main memory databases systems [4, 17, 20, 21, 22, 23]. Bohanon & al present the architecture of a main memory storage manager called Dali. It is a toolkit providing recovery and concurrency control features. Its primary goals are to serve as the lowest level of a database system and to support transaction processing in performance-critical applications. The main feature of the Dali storage manager is the use of a direct access to data in shared memory rather than via inter-process communication, which is relatively slow, but is not portable to a parallel shared nothing environment.

Scalable Distributed Data Structures (SDDS) are another way to use main memories as resident storage for databases [18,20,21]. They are usually based on a dynamic hash function as a partitioning scheme and provide good performances and scalability. In Act21 we use an SDDS method (CTH*) that is order preserving (like range partitioning) and less sensitive to data skew.

Incorporating a semantically rich data model (like object derived models) in a high performance database systems, is very difficult [2]. The transaction model associated (nested model) incur a lot of overhead in concurrency control. The model presented in [10] is such a solution but the transaction programmers are responsible of managing the down-ward inheritance of locks from a transaction to its sub-hierarchy. In our case, we have eliminated the need of controlling the lock transfers by adopting an automatic approach where locks are retained for all the sub-transaction when the holding transaction completes its work. This last waits then for the termination of its sub-hierarchy before signalling its own completion to its ancestor.

One open problem known as to be the most serious limit to parallel MMDBs is the relatively high cost of the atomic commitment protocol for distributed transactions. Park & al, have presented an approach that combines the advantages of the pre-commit and group commit in parallel MMDB while avoiding the consistency problem. The Causal Commit protocol [16] is such an approach. In this case, all redo logs (of all participants in a global transactions) are sent to the coordinator when the request vote is processed, avoiding thus any disk activity in cohorts. The output operation is done by the coordinator to flush the logs to its local disk. This technique is under implementation in our prototype to replace the 2pc approach, where 3 output operations are needed to perform a transaction commitment.

9. Conclusion

In this article, we have presented an approach to build parallel Main Memory DBMS using the concepts of distributed open systems and databases actor. An overview of “Act21” a parallel DBMS being developed at “Institut National d'Informatique”, has been presented briefly.

We have also presented a model of transaction management (nested transaction model) that is suited for our actor like data model. Recovery techniques (fuzzy ping pong checkpointing) are also adapted to the use of “Scalable Distributed Data Structures” SDDS as a storage manager for a parallel MMDB.

We have conducted some preliminaries experiments with the transaction techniques used and the results obtained are very encouraging. We hope to carry out more tests in a near future including long transactions.

Some improvements can be provided, to our architecture, particularly for 2PC extensions to support “group commit” and “pre-commit”. This can be done by keeping in main memory the tail of the global redo log and using techniques like those presented in [16,23] where log records, and some precedence informations are centralized in one site.

References :

[1] Agha, G., ACTORS. a model of concurrent computation in distributed systems, *MIT press 1986.*

[2] Bassiliades, N., Vlahavas, I., PRACTIC : A concurrent object data model for a parallel object-oriented database system, *Information Sciences 86 (1-3), p 149-178, 1995.*

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

[4] Bohannon P., Lieuwen D., Rastogi R., Silberschatz A. and Sudarshan S., The Architecture of the Dali Main Memory Storage Manager, *The Intl. Journal on Multimedia Tools and Applications 4,2, March 1997.*

[5] Bouzeghoub, M., Gardarin, G., Valduriez, P., Les objets, Editions Eyrolles 1997.

[6] Date, C.J., Introduction aux bases de données, International thomson publishing France, 1998.

[7] DeWitt, D., Gray, J., Parallel Database Systems: The Future of High Performance Database Systems, *Communications of the ACM, Vol. 35, No. 6, June 1992*

[8] DeWitt, D., Naughton, J., Shafer, J., Venkataman, Sh., Parallelizing OODBMS traversals: a performance evaluation, *VLDB journal Vol 5 N°1, page 3-18, Jan 1996.*

[9] Garcia-Molina H., Salem K. Main Memory Database Systems: An Overview. *IEEE Trans. Knowl. Data Eng., Vol 4, N°6, pp. 509-516, Dec. 1992.*

[10] Harder T., Rothermel K., Concurrency Control Issues in Nested Transactions, *The VLDB journal, Volume 2, N°1, pp. 39-74, 1993.*

[11] Hewitt, C.E., Viewing Control Structure as Patterns of Passing Messages, *Artificial intelligence, 1977.*

[12] Hidouci W.K., Zegour D.E., Actor Oriented Databases, *WSEAS Transaction on computers, Issue 3, Volume 3, pp. 653-660, July 2004.*

[13] Hidouci, W.K., Zegour D.E., Un SGBD Objet par acteur .*WDAS 2002. Workshop on Distributed Data structures. Paris 21-22 -23 mars 2002*

[14] Jagadish H.V., Silberschatz A, Sudarshan S., Recovering from Main Memory Lapses, *Proceedings of the 19th VLDB, Conf. Dublin, Ireland, 1993.*

[15] Le Gruenwald Le, Jing Huang, Margaret H. Dunham, Jun-lin Lin, Ashley Chaffin Peltier, Survey of Recovery in Main Memory Databases*, Engineering Intelligent Systems 4/3, pp. 177-184, Sept. 1996.*

[16] Lee I., Yeom H. Y., Park T., A New Approach for Distributed Main Memory Database Systems: A Causal Commit Protocol. *IEICE Trans. Inf. & Syst., Vol E87-D, N°1, january 2004.*

[17] Lin J., Dunham M.H., A Survey of Distributed Database Checkpointing. *Distributed and Parallel Databases, 5: 289-319, 1997.*

[18] Litwin W., Sahri S., Implementing SD-SQL server: A Scalable Ditributed Database System, Intl. Workshop on Distributed Data and Structures, WDAS 2004, Carleton Scientific.

[19] Litwin, W., Schwarz, J.E., LH*rs : A high availability scalable distributed data structure using Reed Solomon codes, CERIA Res. Rep. 99-2, Paris 9, 1997.

[20] Litwin, W., Neimat, M. A.., Schneider, D. LH*: A Scalable Distributed Data Structure. CERIA Res. Rep. Nov. 93, Paris 9, 1993.

[21] Litwin, W., Neimat, MA., Schneider, D., RP* : A Family of Order Preserving Scalable Distributed Data Structures. *Proc. Of 2Oth conf. VLDB, chile, 1994.*

[22] Ndiyae Y., Litwin W. and Risch T., Scalable Distributed Data Structures for High-Performance Databases, Tech. Rep, ceria Paris dauphine Univ. 2000.

[23] Park T., Yeom H. Y., A Distributed Group Commit Protocol for Distributed Data Systems. Tech. Rep., PDCS99-GC, Department of computer engineering, Sejong Univ., Korea, 1999.
[24] Yonezawa, A., ABCL: an object -oriented concurrent system, *MIT press, Cambridge MA, 1990.*
[25] Zegour, D.E., Scalable Distributed Compact Trie Hashing. *Inform. Soft. Tech., Elsevier, 2004*.
[26] Zegour, D.E., Adaptation of Trie Hashing for Distributed Environments, *Carleton Scientific, Canada, 2004*