International Journal of Science and Modern Engineering (IJISME)
ISSN: 2319-6386, Volume-1 Issue-9, August 2013

Virtual Differential Storage Based K-Rollback Concurrency Control Algorithm in Distributed Shared Memory Systems

Abhinav Aggarwal, Rupika Srivastava, Sumit Malik, Kirti Meena, Poonam

Abstract*: Most of the algorithms that exist today for concurrency control over distributed shared memory, either fail to provide a scalable model or involve a large communication overhead for establishing consensus over the state of the shared variables. After a thorough study of some of the efficient algorithms this field, this paper introduces a functional view of a holistic approach, which exploits the best features of all others. It provides a virtual differential storage, which allows fast replication and compact storage, along with a strong subversion control over rollbacks in time, which provides better fault tolerance. It also talks of an intelligent logging mechanism, where the read/write records are used actively by the central controller to provide exclusion over Above all, the algorithm is best implemented in LISP or Scheme due to its functional nature. This make the implementation computationally very fast. A trade off, however, exists between the implementation complexity and the quality of the final product.

***Index Terms*— Log, Page, Concurrency, Shared Memory.**

I. INTRODUCTION

As Distributed Operating Systems become more and more common, consistency and concurrency control become issues of contention and importance. Many of the shared resources like Databases, Shared Memory, Log files and the likes, have to be allotted judiciously while avoiding starvation and deadlock among demanding processes. This paper introduces an algorithm, which uses a log based concurrency and consistency maintenance system for enhanced performance using differential disk chaining [1] of the shared memory.

Concurrency control algorithms [1][2][3] are usually classified as locking, timestamp ordering [4][5] and validation (also called optimistic approach [6]). The correctness of a concurrency control protocol is usually based on the concept of serializability [8]. Yoav Raz provides a detailed analysis in his paper on commitment ordering providing the groundwork for concurrency algorithm in his 1992 paper on Commitment Ordering in Databases [10].

Many algorithms have been proposed in past for concurrency control, including some based on distributed control. One such algorithm, for example is Sirius-Delta for database systems developed at INRIA, in which the integrity is maintained by distributed controllers in the presence of concurrent processes.

In Sirius-Delta the cooperation is achieved by the combined principles of atomic actions and unique timestamps. Then we have token forwarding protocols for very large distributed Hierarchical databases, called Hierarchical Token Forwarding Protocol and a commitment control protocol called Multi-level-consistency Protocol [9], by Tao and Williams.

Another popular concurrency control protocol was provided by William Weihl in his 1988 paper on Commutative-Based Concurrency Control for Abstract data Type. He proposed two different algorithms, one using intention lists and other using undo logs [11]. His algorithms are designed for recoverability using two different sets of data structures: post committal Intentions list and pre-committal logs.

Bernstein and Goodman [7] [12] have also shown that another class of algorithms based on Two Phase Locking and atomic actions for databases, which is by far the most commonly used algorithm, and is used with several adaptations to provide faster access and better concurrency control. However, the Two-phase locking algorithm (2PL) as a concurrency control method may restrict the performance of a shared-nothing system more severely than that of a centralized system due to increased lock holding times. Also, in such cases, the deadlock detection and resolution are an added complication. Hence, many variances of it are available which try to improvise as deadlock and starvation free protocols, e.g. Wait-Die, Wait-Depth and Wound-Wait implementations. The proposed algorithm is deadlock and starvation free and tries to provide exclusive access without using locks explicitly.

The shared memory referred to in this paper has been designed to be kept in the virtual address space. The pages are stored in Virtual Hard Disks, mounted on Virtual Machines. These machines run on a central server, governed and managed by a Controller. Virtualization has been chosen as the base of working here so that live replication is easier and copies of the entire chain can be maintained over remote servers for a failover during disaster recovery. We also use Differential Storage over these Virtual Hard Disks so that only the differences in the each section of the memory are stored, from the last updated values of each. A chain of disks allows easy rollback up to K slots in time, hence, the term K-rollback in the name. We also use intelligent and active logs to allow multiple readers and multiple writers concurrent access to the distributed shared memory. However, the writers can only commit changes over the most updated copy of the pages, for which they might have to download the pages at least

Manuscript Received on August, 2013.

Abhinav Aggarwal, Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Roorkee, Roorkee, India.
Rupika Srivastava, Department of Electronics and Computer Engineering, Indian Institute of Technology (IIT) Roorkee, Roorkee, India.
Sumit Malik, Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Roorkee, Roorkee, India.
Kirti Meena, Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Roorkee, Roorkee, India.
Poonam, Department of Computer Science and Engineering, Indian Institute of Technology (IIT) Roorkee, Roorkee, India.

International Journal of Innovative Science and Modern Engineering
IJISME
www.ijisme.org
Exploring Innovation

once before the final commitment. They always store these pages in their local cache, and changes in the cache are not reflected in the global copies.

It is assumed that the shared memory is constructed of pages, and each page is constructed of sectors, and that pages are the minimum units for reading and writing. For an enterprise, the typical sizes of a page and a sector can be around 512MB and 512KB respectively.

This paper does not discuss authentication and authorisation of clients. It is assumed that the users are already authenticated and have authorisation for the pages they are demanding. It focuses entirely on fair grant of requested pages.

II. CONTROLLER

The architecture of the communication between a client process and the Controller is explained through an abstraction provided in Fig. 1.

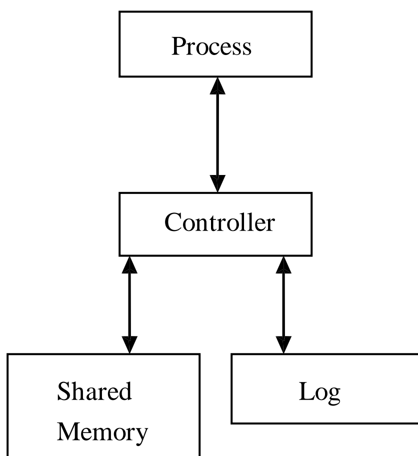

Fig. 1 Architecture of the Server

The algorithm is based on the concept of a centralized server, called the Controller, which interacts with the clients, called Processes, and provides them access to pages in the shared memory. This memory has been implemented in the form of a chain of differentially expanding Virtual Hard Disks (VHD), in such a manner that at any point of time, we have exactly 'K' such disks in the chain. Each disk stores the differences in the pages of shared memory, from their last value stored in the previous set of disks.

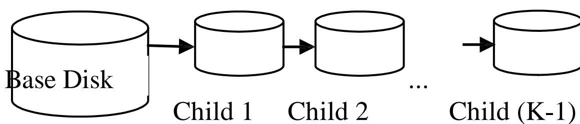

Fig. 2 Differential Chain of the Shared Memory

The Base Disk stores all the pages of the Shared Memory, with the initial data, that will be used by all the clients sending requests (in the form of Tuples, explained later) to the controller. This disk can be dynamically expanding, or fixed, depending on the storage requirements of the organization using it. One may go with the former if it is expected that the number of pages may vary with time. Each of the child disks is a differential disk, which is initially allocated some fixed size, and then dynamically expanded as the requirements increase. We store information only those sectors of each page, which have been modified by the clients. Thus, whenever a client submits an updated page, the controller checks to see what sectors have been modified, and then stores these changes in the disk next in chain to the last disk where the changes were stored. If, however, we run out of disks to store any further changes, we flush the contents of a set of these disks (first few in line) back to the base disk, thus committing these changes in the latter, and shift the contents of the remaining disks to their parent disks. This way, we make space for the new changes, and maintain K versions of each page at all times. It may also happen that each page has its latest set of changes in different child disks, which is not a problem. This way of maintaining shared memory gives us the dual advantage of keeping a storage efficient version control over the contents of the same, as well as allows us to replicate the entire chain over remote servers for disaster recovery. An important point to remember in this scenario is that the computation of changed-sectors (or dirty sectors) and the flush operation over these disks happen in the critical section of the controller, so that no operation can interrupt this. It can either be carried out by a different thread running or a parallel unit in the controller. We assume that the client tuples can still be handled during this period, without interrupting any of the mentioned critical operations. Finally, to decide the optimal number of disks that should be flushed into the base disk when such an operation takes place, it has to be kept in mind that if this number is kept too low, frequent flush operations may be required, and if kept too high, large delays and error chances might creep in. Thus, a good heuristic to decide an optimum value is to track the average write frequency over the pages, and compare it with the average time it takes to apply the differences stored over the base disk.

For each tuple that the Controller receives, it authorizes the client sending the tuple and carries on a set of operations on successful authentication. If the client sends read or update tuple for a set of pages, the Controller checks where the last updated copy of each page (in the set) is stored and uses this information to decide if a flush operation is required or not. Once the decision and a necessary action, if any, has been taken, it reads out a copy of each page from the chain, and sends it to the requesting client, while making an appropriate entry in the log file, discussed in detail in further sections.

The Controller, being centralized, also maintains a small piece of information, called the Log Strip. This strip contains a timestamp and one entry for each page in the shared memory, indicating the client running its exclusive period over the page. All the request tuples received by the Controller are synchronized with the clock according to which the timestamp is computed. This clock resides inside the Controller and may be different from the local clocks at each client. The entries of the Log file are used to find out about the client having an exclusive access over each page. The log file also interacts with this strip to update its later entries.

III. COMMUNICATION PRIMITIVES

It uses message passing communication primitives, and all requests and replies are

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication

tuples composed of six attributes. It uses three types of request tuples: read, write and update and two response tuples from the server: success and abort. A brief explanation of the attributes is:

1. Process Number: The first attribute of every request and reply is the process-ID of the requesting process. It is assumed that all processes are uniquely identifiable on the Distributed System, and it is the unique ID which a client uses to communicate to the server.

2. Page list: The second attribute of the requests and responses is a set of page numbers that a process is demanding to access or is granted an access by the controller.
3. Read Times: The third attribute is the time-stamp at which the read request message was sent, according to the client's local clock. In case of a write request, it is the time at which the client had read the said page earlier, according to the Controller's clock. In the reply tuples, this value is set to the timestamp at which the read request for the set of pages was successfully entered into the log file.
4. Write Times: The time according to local clock in a client when the write command was issued. In success (to write), it is the time when the log entry in the server is written after successful execution of the write request. In success (to update), abort (to write), abort (to update), it is the time of last update of the said page(s).
5. Gestation Period: It is the time-period for which a process requests an exclusive write access to a page. A gestation period is specified for all pages for which the exclusive write access is requested, and if it is not specified, it is assumed that the process is not interested in an exclusive access, and a concurrent non- exclusive access is granted. There is an upper limit to the value this field is allowed, and if a process requests an exclusive write access with a gestation period field larger than that, its request is not granted, and an abort (to read) is issued with the gestation period field set as same as the requested gestation period. In abort (to write) case, it is usually the time left in the exclusive access of the process.
6. Lag: It is the time left before the exclusive access (gestation period) of a process starts, and the time it must wait for, before sending the exclusive write request. It is equal to the sum of all the other processes' gestation periods that have been granted before this request arrived, and any free non-exclusive access in-between, if any. Its use in various messages is listed in the table alongside.

Various messages that are passed are listed as follows:

1) Read: This is the first tuple any client sends to the server.
   Apart from the usual plain read request, it can be the read before a write operation has to be done. The controller does not permit a process to write without having read a page, lest it overwrite some other process's work, hence, every process has to read before write. In such cases, the gestation period is set to the length of the time the process wants an exclusive access to a page. A read request is replied with a success (to read) message, if the request is granted, with gestation period and lag period set. Gestation period of the success message is set to the granted gestation period, which starts at the end of the lag period counted from the time the reply was sent from the server's end. Lag field on read is the time a client is willing to wait before being granted an exclusive write, and is by default equal to zero (which is taken as equal to infinite).
2) Write: A write request is issued by a client at the start of its gestation period. Read-Time and Write-Time for this request are respectively the time returned by the controller on the client's last success (to read) on the pages it is trying to write on and the time this write request was issued by the client. Lag and gestation period are the returned values on success (to read) messages the client had received earlier from the server. A write request is always preceded by an update request.
3) Update: An update request is issued before a write request is issued, so that the client only writes to the most recent copy of the pages. It is issued to see when the page requested was last updated. The read-time entries are the values returned by success (to read) messages, and it can be used to reset gestation period, by specifying a newer value in that field. If possible, the controller allows the client to increase the gestation period, and makes an entry of that in the log, if not, then the abort (to update) is issued, and the client should start all the way over from read to write with newer values. Normally, gestation period, lag, write-time fields would be 0, unless the client wants to have a different gestation period. update is replied with success (to update) messages, which in their page-number attributes contain the pages that need to be updated by the client by issuing a new read request, and gestation-period and lag values are set to the remaining gestation-period and remaining time before the gestation period of the process starts, according to the controller clock. Also, the write-time and read-time fields contain the values the pages were last written and the time they were read respectively. The read in last sentence refers to the read made by the client sending the update request. No write is allowed before a client makes an update call, and an update call must be initiated just before the gestation period so that the process has an updated copy of the page it wants to write over.
4) Success: There are three types of success messages that are sent by the controllers, in reply to read, write and update requests, namely success (to read), success (to write) and success (to update). A success (to read) is sent when a read request is processed successfully. The response tuple has its gestation period equal to the gestation period sent in the read request, its lag period, which is the time it should wait before its gestation period starts, set in the lag field. If the process sent a gestation period in request larger than the maximum allowed value, then an abort (to read) is sent instead, as explained later. Its write-times field is 0, and read-time is the time in the controller's clock when it was processed correctly and a log entry was made in that regard. This time matches the time in log entry regarding this request.

Success (to write): The message issued in response to a

successful write request. The read time in this response is the time the controller had returned for the first read request this client had sent asking for an exclusive access, and further update and write requests. The write-time field is the time when the process succeeded and a log entry was made for this request and the page-number field lists the pages on which the operation was successful. The lag and gestation period fields are by default 0, and the lag field can be used for informing the client about the current lag value on the listed page(s). Note that the gestation period, the period for which the exclusive access was granted expires in case of a successful write, and the remaining period of the exclusive access instead becomes a non-exclusive concurrent access time, in which any process can access the page (concurrent non- exclusive access).

Success (to update): This message is used as a response to update requests. The page-number field is the list of pages for which an update exists, and for those pages, the write-time field lists the time of last updates. Gestation period and lag fields list the remaining gestation and lag periods for the given client.

5) Abort: An abort is issued in case when a request is denied due to some reason. There are three abort messages, each for read, write and update requests.

Abort (to read): For this situation, the page-number field lists the pages for which the read request has failed. In case it is due to the gestation period requested by the process being more than the allowed value, then the gestation period of the abort response is equal to the maximum allowed gestation period on the said page. Also, if the request is denied because its set lag period was not satisfiable, then the lag value for that page is returned in the lag field.

Abort (to write): It is issued when a write request fails. A write request can fail because (i) a process tried to write before its gestation period started or (ii) because it did not make an update request before write request. In both cases, the page-number field contains the pages on which write failed, and the gestation-period field is amount of remaining gestation period, and the lag field contains the remaining lag period. Noticeably, unlike success (to write), abort (to write) does not vacate the exclusive access of the client, and the client retains exclusive access for the full length of the remaining gestation period.

Abort (to update): It is sent for pages on which no updates occurred and do not need be re-read. For this, page- number field contains the list of pages for which no update occurred, and write-time is the time when the last updates were made on the said page(s). In case of these pages, write proceeds without having to call read again.

IV. LOG FILE

The heart of this algorithm is its logging mechanism. The structure of this file is maintained as an ordered set of tuples, called Log tuples. Each tuple has the following attributes:

1. Entry ID: A unique value given to all the tuples in the file, and is used to identify each entry.
2. Time Stamp: The time according to the clock at the Controller, when the tuple was added to the file.
3. Page Number: The page ID of the shared memory for which the log entry has been made.
4. Process Number: The client authentication ID, which is unique for every client that interacts with the server.
5. Access Mode: 'R' for read request, 'W' for write request, 'U' for update request
6. Read Time: The time at the which the request for reading the page was successfully completed. It includes the time when the page was downloaded by the client into its local cache and an acknowledgement was received by the Controller regarding the same.
7. Last Update Time: The time at which the page, specified by Page number, was last updated by any client.
8. Number of Readers: The number of clients that are currently reading the page, specified by the Page number
9. Current Lag: The time for which at least one client is holding an exclusive write access over the specified page.
10. Gestation Period : The exclusive access time requested by the client for the specified page and approved by the Controller.
11. Pointer: The entry ID of the next read entry with non-zero gestation period for the same page.

With these attributes of each tuple stored in the log file, the Controller maintains a dedicated thread to interact with such a system, and does a lot of operations on the entries. Since the last attribute requires a pointer to the next read request, it is only updated when that request arrives. Also, the Current Lag is updated as the clients finish writing and exhaust their gestation periods. Thus, the Controller interacts with each entry of the log, and may update them at any point in time. This renders the log file and active nature, in contrast to the passive logs maintained by most of the other algorithms. To avoid redundancy, the log file contains sufficient attributes to calculate all data required in serializing the write requests over a page, and thus, no extra storage is used to store this information somewhere else. This renders the logging mechanism an intelligence factor, as a lot of information is inferred directly from the contents stored in this file.

For each request that the Controller approves, an update to the log file can be done with the help of the following pseudocode:

```
updateLog (Message M)
if(!validate(M)) { abortMessage(M); return;
}

i = createNewLogEntry(); log[i].TimeStamp := getTime();
log[i].PageNumber := M.pageNumber(); log[i].ProcessNumber :=
M.processNumber();
log[i].AccessMode :=
'R' if incoming request is read(M)
'W' if incoming request is write(M)
'U' if incoming request is update(M)

log[i].ReadTime := log[i].TimeStamp if M.AccessMode = 'R'
|| M.AccessMode = 'U'; M.ReadTime if M.AccessMode = 'W';
if(log[i].AccessMode) == 'W'
     log[i].LUT = log[i].TimeStamp;
else
{

log[i].LUT = 0;
for(j=i;j>0;j--)
{
if(log[j].AccessMode =='W' &&
log[j].PageNumber==log[i].PageNumber)
{
log[i].LUT=log[j].TimeStamp;
break;
}
}
}

if(log[i].AccessMode == 'W')
for(j=i-1; j>0; j--)
```

```
if(log[j].PageNumber == log[i].PageNumber)
log[i].NumberOfReaders = log[j].NumberOfReaders-1; break;

else  if(log[i]AccessMode == 'R')
for(j=i-1; j>0; j--)
if(log[j].PageNumber == log[i].PageNumber)
log[i].NumberOfReaders =
log[j].NumberOfReaders;
break;

log[i].Current_Lag := ComputeLag(M.processNumber,
M.pageNumber);

log[i].GP := max (M.GP, maxGP);

log[i].Pointer := NULL;

for(j=i-1; j>0; j++)
if(log[j].PageNumber == log[i].PageNumber)
if(log[j].AccessMode = 'R')
log[j].Pointer = i; break;

successMessage(M, i);

}
```

The log strip used by the Controller works in strict synchronization with the log file. It maintains a virtual stack of all the clients currently lined up in the request for exclusive rights over a particular page, so that whenever a client is done with its critical work, the values in this strip can be updated. Moreover, these values are further used by the Controller to update the Current Lag attribute in the log file. Thus, a strong feedback mechanism exists between the log file and the log strips. Care must be taken to avoid any errors that might creep in.

V. LOG STRIPS

As mentioned earlier, these strips are used by the Controller to maintain information about the clients which are currently running their gestation periods over each page in the shared memory. To maintain this information, a stack of the pending clients is maintained for each page. The log file entries are used as stack entries here, thus, eliminating any extra space that may be required for the same. Consider the ordered subset of all tuples in the log file which correspond to read requests to a common page, say P. This subset, then, forms the stack required for updating information in the strip. As an example, if the entry corresponding to Page P in the strip is, say, C, then the local clock at the Controller is checked to see when the gestation period for C is over, and it really expires, the entry corresponding to C in the log file is checked for the pointer to the next client waiting in line. This next client's entry is then written over the existing entry for P, and the same process continues.

Using the log strips, the Controller updates the Current Lag field in the log file using the following sequence of steps:

```
computeLag(ProcessNum, PageNum)
{
if (ProcessNum == LOG_STRIP.Page(PageNum))
return 0;

lag := stack[top].Gestation_Period - LOG_STRIP.clock;

for(i = top-1; i>=0; i--)
if (stack[i].ProcessNumber == ProcessNum)
return lag;
else
lag += stack[i].Gestation_Period;
}
```

A quick look at the above pseudocode clearly explains the basic steps required for the log file to update its entries in the lag option.

With such synchronization features, the Controller always maintains a proper exclusion over the write requests to a particular page.

VI. REQUEST PROCESSING

Whenever a client wants to make some change, it makes a request to the controller. The controller, if can satisfy the request, it grants access to the client, and lets it process, and makes an entry for it in the log, and sends a success message. A client can do two things, it can either want to read a page in shared memory, or it may want to modify it. In first case, the client should make a read request with gestation period field 0 and lag field 0. In that case, if there is no other process that has exclusive access, it is granted read access. If the page is in control of a process in the middle of its gestation period, an Abort (to read) is sent, and in the lag field of the response is the time for which the client must wait before reading again.

In case of write, the process first sends the read request with a gestation period field equal to the gestation period field required by the client. The controller replies with a success (to read) field if the request can be granted in the conditions specified by the client, and the lag field in the reply is the time it would have to wait before write. If the process specified a gestation period too large, or if the request cannot be specified in before the maximum lag the client specified, then the abort (to read) is sent with gestation period field equal to the maximum allowed value of gestation period, and lag equal to the current lag value on the requested page.

After a successful read, as the time the client must wait (lag) comes closer to zero, the client must send an update request to see if the value of the pages on which the client plans to modify have been overwritten in the time it has waited or not. If the client receives success (to update), it must re-read the page before writing. If it receives abort (to update) instead, it implies that the pages have not been changed meanwhile and it is alright to update them without having to read them.

When a client receives a read request for a page with gestation period field non-zero, it calculates the current lag on that page, if that value is greater than permissible lag as specified by the process in its read request, it replies with an abort(to read with lag field filled with current lag). Else if the gestation period specified is larger than the maximum allowed value of gestation period, an abort (to read) message with maximum allowed value of gestation period set in 'gestation period' field is returned.

If both of these conditions are not violated, then the controller goes through the log to find out if any process is currently in its gestation period over that page. It starts from the starting point of the log, and finds out the first entry about the given page number. It checks if that entry was "r" (read) or "w" (write). In "r" entries with 'gestation-period' > 0, if 'lag' + 'time-stamp' + 'gestation period' – current time > 0, then, this client might have an exclusive access over that page. Hence, we jump checking from one entry to another using 'pointer' field to skip unrelated entries, to find if there is any such process. If there is indeed any such process, then we look if there is an entry for that page with 'process-number' which is in "w" mode or not. Because a process loses exclusive control after it has successfully updated an entry,

if there is any such entry after time 'time-stamp' + 'lag' of the original "r" entry that entailed the process the exclusive access.

If there is no "w" entry for the process, that means that that process holds an exclusive access pass over the requested page (or, that we are in the gestation period of that process's request) and an abort (to read) is returned to the client. If not so, and there exists a "w" entry, then it is taken that the process has already written, its exclusive access is assumed expired, and a success (to read) with lag = current lag is returned.

During write, the controller follows similar procedure. The first step is to see if there is an "r" entry with the same "read-time" as the arriving write request. If there is, we check for that entry if 'current lag' + 'gestation-period' – current time <0. If it is so, then we are in the gestation period of that client. Then we check for the latest "r" entry in log for that page and process, and if there are any "w" entries after that. If there is no other "r" entry apart from the one with same "read-time" as the write request, we know that the process has not updated its copy of the page before writing, and an abort (to write) is sent as reply, with gestation-period field= remaining gestation-period of the process. If there is an "r" matching with the process- number of the given process, and there are no other "w" entries about that page-number, we assume that the process has and updated copy of the page, and we allow it to write, a log entry of "w" mode with process-number of the requesting process and requested page-number is made in the log, and an entry about it is made in the differential disk.

If we are not in the gestation period of the requesting process, an abort (to write) is sent instead.

When a process receives an update request with a 'read-time' field = rd_time, say, we see if there are any entries in log for the given page and process with mode = "r". If there is, we check if the given page has any "w" mode entries after that . If there are, we send a success (to update) to the sending client otherwise we send an abort (to update). The process is then expected to read the page again before issuing a write() request.

So, following messages are transferred **for read:**

Client: read()
Controller: success (to read)
Client can now read

Client: read() Controller: abort(to read)(. . . lag)
Client should try read after "lag" amount of time

For Write:

Client: read(,,,0,gest_pd, permissible_lag)
Controller: success (to read)(,,,rd_time, gest_pd,lag)
Client waits for "lag" amount of time, then sends an update request, and waits for the reply (abort, or success)

Client: update (,,,rd_time,0,0) Controller: abort (to update)(,,,,,) Client: write (,,,rd_time,,)

Another scenario is:

Client: read (,,,,gest_pd, permissible_lag)
Controller: abort (to read)(,,,rd_time, max_gest_pd, 0)
Request denied due to gestation period requested was larger than the maximum allowed value

Client: read (,,,,gest_pd, permissible_lag)
Controller: abort (to read),,,rd_time, 0, lag)
Request denied due to the current lag on the page is greater than the permissible lag the request had mentioned.

Client: read(,,,gest_pd, permissible_lag)
Controller: success (,,, gest_pd, lag)
Client waits for "lag" amount of time, sends an update request

Client: update(,,,rd_time, 0,0)
Controller: success (to update) (,,rd_time, lst_up, rem_gest, rem_lag)
Client should now read the said page before writing. However, it still has the access to the page.

VII. STARVATION AND DEADLOCK

There are four different conditions called Coffman Conditions; that have to be satisfied for deadlocks: mutual exclusion, hold-and-wait, no preemption and circular wait. Violation of any one condition is enough to prove that an algorithm is deadlock free. The proposed algorithm allows exclusive access to resources and does not pre-empt the exclusive access of a process to a resource, but provides a violation of the hold-and-wait condition, and hence, is deadlock free, as explained below.

The exclusive access to a resource is granted based on a request made such by the process. The time period, called Gestation Period; during which a process is granted exclusive write access to a page in the memory is always known before- hand and limited to a maximum value allowed for that particular controller, hence the process is not granted an exclusive access forever. Thus, it cannot go in hold-and-wait situation. Sooner or later, whenever the process's gestation period expires, it will have to give up the exclusive rights and other processes will get the shared resource. This arrangement can never spiral out into an infinite wait, and hence, cannot go into a deadlock. The time slots allotted for access of a particular process are decided pure based on first-come-first- serve (FCFS) policy. The process is allotted the access for the duration it is demanded (Gestation Period) at the end of already allotted time-sequence (a lag), or if it is possible; in the time intervals which were otherwise allotted to some other processes of which the access has since been revoked. Hence, the process always has a fixed time before which it will be granted all the requested rights, and hence starvation cannot occur. It is a direct consequence of allotment based on a fair policy of FCFS unlike prioritized scheduling; and at the time of receipt of a request instead of doing it later. Also, non-exclusive concurrent access are always granted to processes who request for non-exclusive access during the time there is no exclusive access granted.

International Journal of Innovative Science and Modern Engineering
IJISME
www.ijisme.org
Exploring Innovation

VIII. CONCLUSION

We have proposed an algorithm on "K-rollback Virtual Differential Storage based Concurrency Control in Distributed Shared Memory Systems", which focuses on concurrency control in distributed shared memory (DSM) environment. The algorithm successfully works on multiple readers – multiple writers scenario. It is free of deadlock as well as starvation. This algorithm is extendible and applicable to several other shared-resources scenario like distributed databases where concurrency control is of major concern. The system allows multiple reads and writes, and the changes are stored in differential disks. This feature allows easy recoverability in case of a crash and can be very useful in systems which keep logs of changes made on the original storage for records and easy rollback. The records in differential disk are flushed pre-decided checkpoints and the original disk is updated based on differential disk.

This algorithm, using a simple and uniform message structure utilizes the modularity of the attributes instead of using different structures for different functions. The pseudo-code provided is based on lisp, but can be modified for any language. It is simple to understand and implement. The algorithm uses log entries for access control and concurrency control, and does not utilize any elaborate data structures like queues and stacks, which is intentionally avoided to keep the overhead low. Log entries are central to algorithm's implementation.

This algorithms relies on good-faith behavior of processes to not to try to hog resources. It is fair and balanced to provide as much access to as many processes as demanded without any prioritizing criterion among the requests. Hence, it is susceptible to cases where the process demand undue large amount of resources for long periods of time. Also, it expects the processes to declare the amount of gestation period (time of exclusive access) before access is granted. This, although prevents deadlock and starvation, can be a tricky for the processes to guess, and most processes, assuming the worst case scenarios would be poised to give out the larges possible values of Gestation period. This problem is further compounded when simultaneous access to several resources is required.

REFERENCES

1. Tanenbaum A. S., Robert Van Renesse, "Distributed Operating Systems", ACM Computing Surverys, vol. 17, no. 4, pp. 419-471, Dec. 1985
2. G. LeLann, "Algorithms for distributed data-sharing systems which use tickets," in Proc.
3. R. H. Thomas, "A majority consensus approach on concurrency control for multiple copy databases," ACM Trans. Database Syst., vol. 4, no. 2, pp. 180-209, June 1979.
4. D. J. Rosenkrantz, R. E. Stearns, and P.M. Lewis, "System level concurency control for distributed database systems," ACM Trans. Database Syst., vol. 3, no. 2, June 1978.
5. Pei-Jyun Leu & B Bhargava, "Multidimension Timestamp Protocols for Concurrency Control", IEEE Transations on Software Engineering, Vol. SE-13, No. 12, December 1987
6. H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," ACM Trans. Database Syst., vol. 6, no. 2, June 1981.
7. P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," ACM Comput. Surveys, vol. 13, no. 2, June 1981.
8. Jiwu Tao, J. G. Williams, Concurrency Control and Data Replication Strategies for Large-scale and Wide-distributed Databases, ISBN: 0-7695-0996-7/01, ©IEEE 2001
9. Yoav Raz, "The Principle of Commitment Ordering, or Guuaranteeing Serializability in a Heterogeneous Environment of Multiple Autonomous Resource Managers Using Commitment", Proceedings of the 18th VLDB Conference, Vancouver, Canada 1992.
10. William E. Weihl, "Commutavity-Based Concurrency Control for Abstract Data Types", IEEE Transactions on Computers, Vol 37, No. 12, December 1992.
11. P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr., "Concurrency control in a system for distributed databases (SDD-1)", ACM Trans. Database Syst., vol. 5, no. 1, pp. 18-25, Mar. 1980.
12. Managing Virtual Hard Disks Using Differencing Disks, Microsoft
13. Corporation, available at: http://technet.microsoft.com/en-us/library/cc720381(v=ws.10).aspx

AUTHOR PROFILE

Abhinav Aggarwal is an undergraduate student at Indian Institute of Technology (IIT) Roorkee in Computer Science and Engineering, with speacialisation in Information Technology, to Graduate with a B. Tech. and M. Tech. Dual Degree in 2014.

Rupika Srivastava graduated from Indian Institute of Technology (IIT) Roorkee with a B. Tech. In Computer Science and Engineering in 2013. She is currently an employee with Samsung India Software Operations.

Sumit Malik is an undergraduate student at Indian Institute of Technology (IIT) Roorkee in Computer Science and Engineering, with speacialisation in Information Technology, to Graduate with a B. Tech. and M. Tech. Dual Degree in 2014.

Kirti Meena is an undergraduate student at Indian Institute of Technology (IIT) Roorkee in Computer Science and Engineering, with speacialisation in Information Technology, to Graduate with a B. Tech. and M. Tech. Dual Degree in 2014.

Poonam is an undergraduate student at Indian Institute of Technology (IIT) Roorkee in Computer Science and Engineering, with speacialisation in Information Technology, to Graduate with a B. Tech. and M. Tech. Dual Degree in 2014.