International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 2 Issue: 11 3352 – 3359

Transactions Processing Subsystems for Databases Based On ARIES Write-Ahead Logging for The Client-Server Architecture Approach

[1]Dominic Damoah, [2] Dr. J. B. Hayfron-Acquah, [3]Edward D. Ansong, [4]Selby L. Maxwell Jnr., [5]Shamo Sebastian
[1345]Department of Computer Science
Valley View University
Oyibi, Ghana
[2]Department of Computer, Kwame Nkrumah University of Science and Technology
Kumasi, Ghana
Contact Email: kwddamoah@gmail.com@yahoo.com

***Abstract*—** This paper proposes a formal framework specification that applies an advanced recovery mechanism, functional in a client-server architecture while addressing atomicity and consistency issues. Another palpable issue in using such dominant architectures is recovery. This paper also addresses this issue in context with the client-server architecture using extensions of the original ARIES algorithm and concepts of Software Transaction Memory. This novelty has been successfully implemented and tested for propriety and applicability.

Keyword; *ARIES, databases; recovery; processing, transaction, logging, write-ahead, Concurrency, atomicity;*

*****

I. INTRODUCTION

Transaction management has gradually become an essential component of database management systems. It enables multiple users to access the database concurrently while preserving transactional properties such as atomicity, consistency, isolation, and durability. With this in mind, it is evident that concurrency is an important feature of transaction management. Issues with locks arise when dealing with accesses to specific portions of a database concurrently by multiple clients. Also, the recovery techniques that apply to customary query-shipping processing would not be appropriate when it comes to recovery at workstations that are a part of the client-server architecture. The proposed framework brings to light how data-shipping mechanisms can be applied to ARIES in the client-server environment to allow flexibility in the interaction between clients and a server and simultaneously confronting issues of recovery at both ends of the architecture.

II. ARIES

ARIES is a fairly recent refinement of the Write-Ahead-Logging (WAL) protocol. The WAL protocol enables the use of a STEAL/NO FORCE buffer management policy, which means that pages on stable storage can be overwritten at any time and that data pages do not need to be forced to disk in order to commit a transaction. As with other WAL implementations, each page in the database contains a Log Sequence Number (LSN) which uniquely identifies the log record for the latest update which was applied to the page. This log sequence number (LSN) (referred to as the pageLSN) is used during recovery to determine whether or not an update for a page must be redone. LSN information is also used to determine the point in the log from which the Redo pass must commence during restart from a system crash. LSNs are often implemented using the physical address of the log record in the log to enable the efficient location of a log record given its LSN.

Much of the power of the ARIES algorithm is due to its Redo paradigm of repeating history, in which it redoes updates for all transactions — including those that will eventually be undone. Repeating history greatly simplifies the implementation of fine grained locking and the use of logical undo operations as shown in [12]. The resulting simplicity allows ARIES to be adapted for use in many computing environments.

ARIES uses a three phase algorithm for restart recovery. The Analysis pass is the initial phase, which scans the log forward from the most recent checkpoint. This pass determines information about dirty pages and active transactions that would be used in the passes that follow. The second is the Redo pass, in which history is repeated by processing the log forward from the earliest log record that could require redo, thus insuring that all logged operations have been applied. The third pass is the Undo pass. This pass proceeds backwards from the end of the log, removing from the database the effects of all transactions that had not committed at the time of the crash.

III. WRITE-AHEAD LOGGING

This is a recovery mechanism where sub transactions in a transaction are not immediately written to disk as they are executed. That is, the final values are can re-written by new logged values. The old values are known us before image (BFIM) and the new values are known as After Image (AFIM) [3].

The recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk.

Write-ahead logging is necessary to be able to UNDO the operation if this is required during recovery.

A REDO-type log entry includes the new value (AFIM) of the item written by the operation since this is needed to redo the effect of the operation from the log.

The UNDO-type log entries include the old value (BFIM) of the item since this is needed to undo the effect of the operation from the log.

If a cache buffer page updated by a transaction cannot be written to disk before the transaction commits, the recovery method is called a no-steal approach.

On the other hand, if the recovery protocol allows writing an updated buffer before the transaction commits, it is called steal.

The no-steal rule means that UNDO will never be needed during recovery, since a committed transaction will not have any of its updates on disk before it commits.

If all pages updated by a transaction are immediately written to disk before the transaction commits, this technique is referred to as FORCE. Otherwise, it is termed NO-FORCE. The force rule means that REDO will never be needed during recovery, since any committed transaction will have all its updates on disk before it is committed.

The advantage of STEAL is that it avoids the need for a very large buffer space to store all updated pages in memory. The advantage of no-force is that an updated page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the input and output cost to write that page multiple times to disk, and possibly to have to read it again from disk [3].

IV. ACID PROPERTIES OF DATABASES

A transaction, exemplified in database management systems, is an execution of a user program, entailing a series of read and write operations. The following are properties that a database management system should adhere to when handling transactions in order to maintain data when concurrent access and system failures come to play:

1. Users should be able to regard the execution of each transaction as atomic.
2. Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database.
3. Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons.
4. Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk.

These four properties are Atomicity, Consistency, Isolation, and Durability respectively.

Users are responsible for ensuring transaction consistency. That is, the user who submits a transaction must ensure that, when run to completion by itself against an instance of a consistent database, the transaction will leave the database in a 'consistent' state. Let us take a look at this scenario, fund transfers between bank accounts should not change the total amount of money in the accounts. When transferring money from one account to another, a transaction should not only debit one account, temporarily leaving the database in an inconsistent state. The user's understanding of a consistent database is maintained when the second account is credited with the transferred amount. Some call this the all or nothing property. The isolation property ensures that even though a problem may occur in the transfer process it enforces that each process is executed separately from the other in a particular manner or sequence [3].

V. DATA RECOVERY

As with almost all complex forms of computer hardware and software, there is always the possibility of failure. It is important for database administrators to adopt effective recovery mechanisms that can recover database contents which have been damaged or lost in times of disasters. Recovery is not an easy process. In some cases it is impossible to totally recovery data that has been lost or damaged. The volatility of memory and timing and complexity of any CPU limit database administrators to precisely reconstruct data [4].

At all times, there are threats to data security, especially when a database or critical transactions fail. These threats include accidental losses attributable to human error, software failure, hardware failure, theft and fraud, improper data access, loss of privacy (personal data), loss of confidentiality (corporate data), loss of data integrity, loss of customers, loss of corporate integrity, loss of availability (through sabotage, for example), exposure through com links, aborted transactions, incorrect data, system failure (database intact), loss of transfers and backups, loss of money, loss of time, and database destruction.

To prevent some of these issues, most corporations and companies using databases have backup and recovery systems. They include, but are not limited to, backup facilities, journalizing facilities, transaction logs (time, records, and input values), database change logs (before & after images), checkpoint facilities, recovery managers, and a restart point after a failure. It is wise that despite having soft-copies of data hard-copies are also very important

VI. CONCURRENCY CONTROL

Concurrency control is a concept that is used to address conflicts with the simultaneous accessing or altering of data that can occur with a multiple user system. Concurrency control, when applied to a transaction processing, is meant to coordinate simultaneous transactions while preserving data integrity [5].

To illustrate the concept of concurrency control, consider two business men who go log onto an online ticketing booth at the same time to purchase a plane ticket to the same destination on the same plane. There's only one seat left to be accommodated, but without concurrency control, it's possible that both business men will end up purchasing a ticket for that one seat. However, with concurrency control, the database wouldn't allow this to happen. Both business men would still be able to access the plane seating database, but concurrency

control would preserve data accuracy and allow only one traveler to purchase the seat.

VII. LOCKS WITH ARIES

The concept of ARIES implements optimistic locking of the sort such that all users are granted the access to the data entity but any change, be it an addition or deletion, are first of all logged to a file. This eliminates the problems facing both optimistic and pessimistic locking. Furthermore, after these changes have been made, the database administrator or programmer can create "rules" by which the database would then be updated based on these log file (These rules are dependent on the size of the database). ARIES adopts a concurrency control mechanism known as basic time stamping. This method doesn't use locks to control concurrency, so it is impossible for deadlock to occur. According to this method a unique timestamp is assigned to each transaction, usually showing when it was started. This effectively allows an age to be assigned to transactions and an order to be assigned. Data items have both a read-timestamp and a write-timestamp. These timestamps would be updated each time the data item is read or updated respectively [6].

Adhering to the rules of the basic time stamping process allows the transactions to be serialized and a chronological schedule of transactions can then be created and logged.

Time stamping may not be practical in the case of larger databases with high levels of transactions. A large amount of storage space would have to be dedicated to storing the timestamps in these cases [13].

VIII. RECOVERY WITH ARIES

There are two general approaches to recovery: the write-ahead Logging (WAL) approach [7] and the shadow-page technique [8, 7].

WAL is the recovery method of choice in most systems, even though the shadow-page technique of System R is used in some systems, possibly in a limited form (e.g., for managing long fields or BLOBs). In WAL systems, an updated page is written back to the same disk location from which it was read. That is, in-place updating is done on disk. The WAL protocol asserts that the log records representing changes to some data must already be on stable storage before the changed data is allowed to replace the previous version of that data on disk. Each log record is assigned, by the log manager, a unique log sequence number (LSN) at the time the record is written to the log. The Log sequence numbers are assigned in ascending sequence. Typically, they are the logical addresses of the corresponding log records [9]. At times, version numbers or timestamps are also used as LSNs [10, 11]. On finishing the logging of an update to a page, in many systems whose recovery is based on WAL, the LSN of the log record corresponding to the latest update to the page is placed in a field in the page header. Hence, knowing the LSN of a page allows the system to correlate the state of the page with respect to those logged updates relating to that page. That is, at the time of recovery, given a log record, the LSN of the database page referenced in the log record and the LSN of the log record can be compared to determine unambiguously whether or not that log record's update is already reflected in that page. The buffer manager, in order to enforce the WAL protocol, uses the LSN associated with a modified page to ensure that the log has been forced to disk up to that Log Sequence Numbers before it writes that page to disk. With the shadow-page technique, as it is implemented in System R and SQL/DS, the first time a logical page is modified after a checkpoint, a new physical page is associated with it on disk. Later, when the page (the current version) is written to disk, it is written to the new location. The obsolete physical page associated with the logical page is not discarded until the next checkpoint is reached. Restart recovery occurs from the shadow version of the page if a system failure should occur. With shadow paging, checkpoints tend to be very expensive and disruptive. This is because a checkpoint is taken only when all activities in the data manager have been quiesced to an action-consistent state. After quiescing, all the modified pages in the buffer pool and the log are written to disk. Then, the shadow version is discarded and the current version is also made the new shadow version. As a result of all these synchronous actions by the check pointing process, restart recovery always happens from the internally consistent, shadow version of the database. Even when the shadow-page technique is used for recovery, logging of updates is still performed. The WAL approach has commercially been much more widely adopted than the shadow-page technique. Detailed comparative analysis between the two methods are given in [7]. In this research, when we talk recovery methods, it is solely based on Write Ahead Logging. The concurrency protocols that we discuss are applicable also to systems that use the shadow-page technique. In the following, we will summarize the original.

IX. THE CLIENT-SERVER ARCHITECTURE

Here we addresses the correctness and performance issues that arise when implementing logging and crash recovery in a client-server environment.

These problems result from two characteristics of page-server systems:

- The fact that data is modified and cached in client database buffers that are not accessible by the server.
- The performance and cost tradeoffs that are inherent in a client-server environment.

We describe a recovery system that we would implement for particular client-server systems. This implementation would support efficient buffer management policies, allow flexibility in the interaction between clients and the server, and reduces the load on the server by performing much of the work involved in generating log records at clients.

The proposed mechanism is a data-shipping system which employs a client-server architecture. The implementation of recovery in this instance involves two main components.

The logging subsystem manages and provides access to an append-only log on stable storage. The recovery subsystem uses the information in the log to provide transaction rollback and system restart. The implementation of recovery also involves close cooperation with the buffer manager and the lock manager. The recovery algorithm is based on original ARIES Algorithm. ARIES is generally accepted because of its simplicity and flexibility features, its ability to support the efficient STEAL/NO FORCE buffer management policy [14],

 ISSN: 2321-8169
3352 – 3359

its support for savepoints and nested-top-level actions, and its ability to support fine-grained concurrency control and logical Undo. However, the algorithm as specified in [12] cannot be precisely implemented in a client-server architecture because the architecture violates some assumptions upon which the original ARIES algorithm is based.

We also describe the recovery manager, paying particular attention to the modifications to the ARIES method that were required due to both the correctness and efficiency concerns of recovery in a client-server system.

It should be noted that the ARIES algorithm has recently been extended in ways that are similar to some of the extensions described in this paper.

X. ARCHITECTURE OVERVIEW

Figure 1 below shows the proposed client-server architecture. This design was driven by the anticipated capabilities, performance, and reliability characteristics of the clients, server, and the network; as exemplified in our everyday object-oriented DBMSs. Obviously, a server is expected to have more CPU power, more disk capacity, and more memory than a single client, but the combined processing power and memory of individual clients is overwhelmingly greater than the server in instances where you have a wholesome number of clients on a network. Clients are expected to be less reliable than the server and may not have all of their resources available for use by the database system. The main cost of communication is expected to be the CPU overhead of sending and receiving messages. The system will comprise two sections: the client's collective archive of methods, which would be associated to user's application, and the server running an independent process.

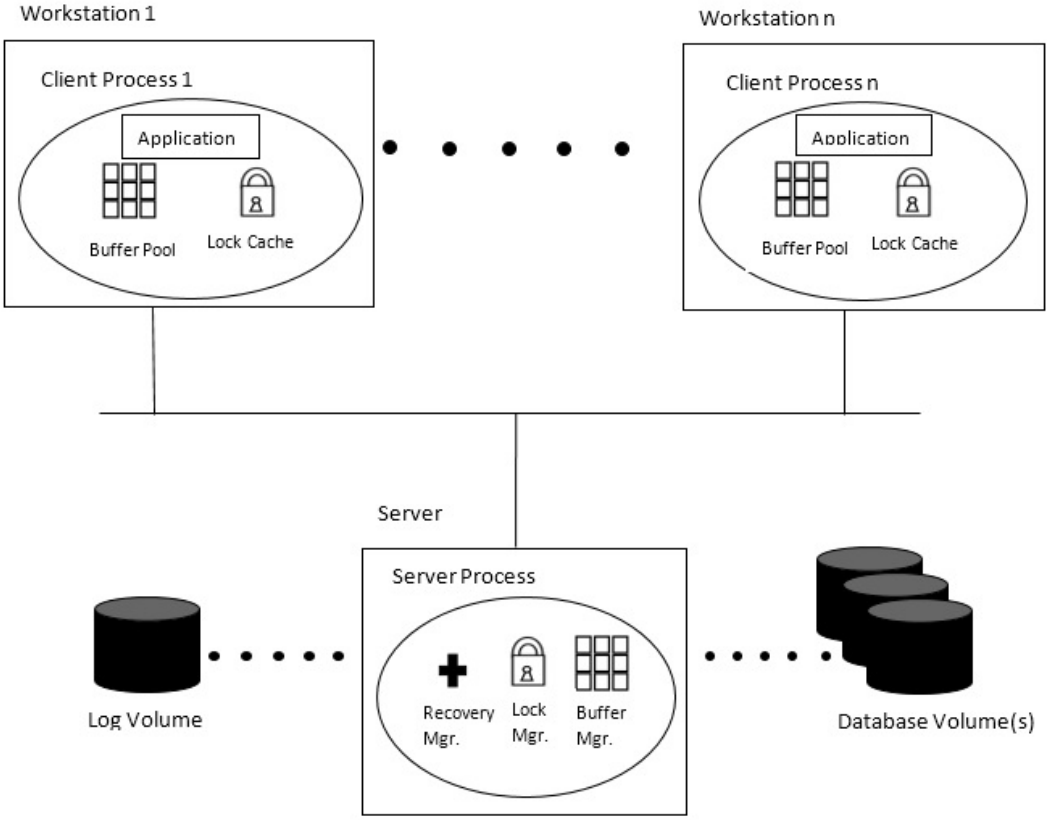

Figure 1. Client Server Architecture for proposed framework

The architecture has a clear division of logging labor between the server and clients. The server hosts the database and a single log volume and also provides support for lock management using software transaction memories (programming without locks), page allocation and de-allocation, and recovery/rollback. Clients perform all data and index changes during normal database transactions. Each client process has its own storage structure (buffer) and transaction memory and runs a single transaction at a time. The server is multi-threaded and allows it to receive requests from multiple clients concurrently. The server uses a separate disk processes so it can perform asynchronous input and output. The system would not support access to multiple servers or federated databases from a single client. We should however note that while Figure 1 shows clients and the server executing on separate machines, it is also possible, for development purposes, to run the server and any number of clients as separate processes on the same machine. Communication between clients and the server can be implemented using reliable TCP connections or UNIX sockets. All

communication is initiated by the client and is responded to by the server. That is, the server responds to requests from the client for specific pages, locks, and transaction services. There should be no means for the server to initiate contact with a client. This would require client to also be multi-threaded increasing complexity of their composure.

As stated above, this mechanism would employ a client-server architecture in which the client sends requests for data and index pages to the centralized server. During a transaction, clients cached received data and index pages in their local storage structure.

Before committing a transaction, the client sends all the pages modified by the transaction to the server. A client's cache is purged upon the completion of a transaction.

Clients start transactions by sending a start transaction message to the server and can request the commit or abort of a transaction by sending another message to the server. The server can then decide to abort a transaction when it faces an error. When a transaction is aborted, the server notifies the client that the transaction has been aborted in response to the next message the client sends to it. This sequence takes place because there is no mechanism for the server to initiate contact with the client. While a transaction is executing the client generates logs based on all updates that are being made to data and index pages. The server receives these logs from the clients in a methodical manner and can abort a transaction when it running out of log space.

XI. CONCURENCY ISSUES

Dealing with problems that have to do with concurrency, worst case scenarios can transpire when multiple clients attempt to gain access to the same data pages. In such situations, we make very good use of transaction memories and buffer managers. When a client is given control to specific data pages, other requesting clients are given controlled access using timestamps which are managed by the server process. During the periods in which clients retain control, sub transactions, if any, inherit locks downwards giving them priority over other requests. After the modified data pages have been logged and flushed to permanent storage, control can now be passed on to the next client according to the servers managed timestamp.

XII. WHY CENTRALIZED DATABASES WHEN APPLYING ARIES IN A CLIENT SERVER ENVIRONMENT?

Most databases are physically located at one place and are managed by one computer. These databases are referred to as centralized databases.

Alternatively, with distributed databases, data are stored in different settings and on different computers and indexes are kept at a central computer to trace the location of data in different places.

An information system manager must decide whether to use centralized or de-centralized databases. There are many reasons why a decentralized database will be selected over a centralized one.

De-centralized or distributed databases are known to be more flexible and permit a number of different units to update and maintain their own data. At the same time, this increased flexibility runs the risk that some units may institute changes that may make them less accessible by others.

It will seem astute to any database professional to manage distributed data with different levels of transparency like network transparency, fragmentation transparency, and replication transparency. With decentralized databases there is an increase in reliability and availability. There is also easier expansion. Decentralized databases have their downfalls too.

Decentralized databases exchange files and therefore may exchange corrupted files or viruses that may affect the entire system. Security of these databases are however difficult to maintain. In decentralized databases the type of data to be exchanged, the process of addressing the data, and the protocol for updating the data must be agreed upon ahead of time and plans must be in place for updating the process.

Having noted all this, we find out that questions still arise concerning the reasons why we choose to apply the proposed mechanism in centralized databases. Let us consider a few factors.

A con of distributed databases is that, aside keeping the addresses of where to locate data, an audit trail is required indicating who updates or retrieves data. With centralized databases we have a controlled "audit trail" because everything is stored in one location. Errors are easily pinpointed and we understand clearly where the confidentiality of a system breaks down. We also need not worry about the difficulty in auditing when computers receiving data increases.

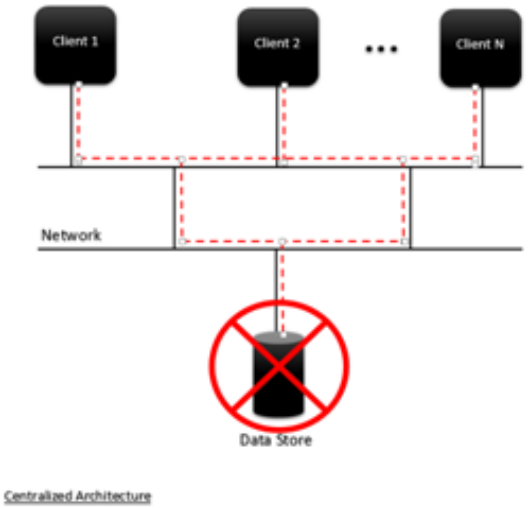

Figure 2. Centralized Architecture without ARIES

With decentralized databases, burdensome procedures are needed to determine the quality of data.

Now our argument concerning the justification of this research is that, in centralized databases lack of backup, inadequate backup, and improper recovery mechanisms may result to complete loss of data while in distributed data systems data loss is limited to nodes affected.

ISSN: 2321-8169
3352 – 3359

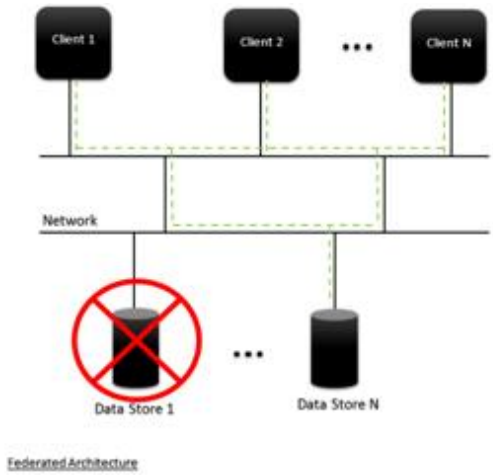

Figure 3. Federated Architecture without ARIES

Therefore to maintain the integrity of centralized systems it is vital to tackle issues that have to do with ACID properties of databases, hence the significance our research. The use of ARIES WAL ensure the continuity of data access even in the course of system failure.

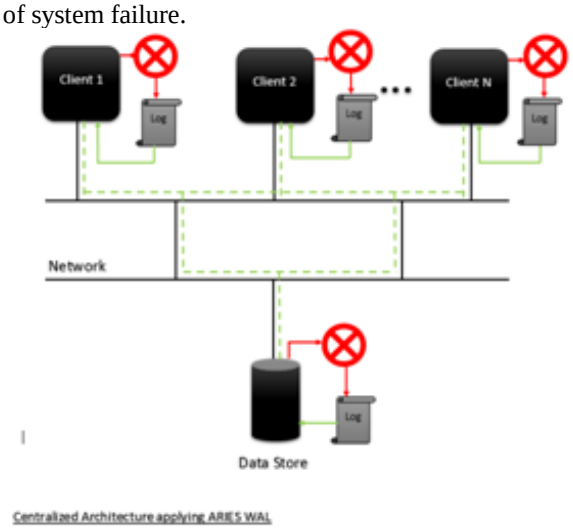

Figure 4. Centralized Architecture with ARIES

Extending ARIES to the client machine further reassures us that even when an individual client encounters a failure during a transaction we can recover it to a consistent state to continue from a certain point without starting all over.

XIII. IMPLEMENTATION

This section talks about the steps or processes through which log are created and sent to the servers circular buffer of logs, before they are flush to the server's permanent storage. The proposed ARIES framework is developed as a log creation, live logging and log transfer mechanism. Its application is run as multiple processes including a centralized server and independent clients on the same system.

A. CREATING LOGS

In creating logs at the client side of the architecture, it is important to follow a systematic file naming convention to indicate the order in which log files are create. The log sequence number (LSN) (referred to as the pageLSN) used to create our logs is used during recovery to determine whether or not an update for a page must be redone. LSN information is also used to determine the point in the log (in our case the log file) from which the Redo pass must commence during restart from a system crash. LSNs are often implemented using the physical address of the log record in the log to enable the efficient location of a log record given its LSN.

```
Home.filepathtolog = FileName;
using (System.IO.File.Create(@"C:\Users\
Projects\ARIES (application)\ARIES (application)\Logs\" + logsequencenumberstring + ".txt")) { }
LogTabPage Page = new LogTabPage(FileName, Suffix);
MainTabControl.TabPages.Add(Page);
MainTabControl.SelectedTab = Page;
//Subscribe to the changed event
Page.Watcher.Changed += new FileSystemEventHandler(Watcher_Changed);
checker = "success";
Page.Select();
globalClientLogToBeSentToserver = logsequencenumberstring + ".txt";
```

Figure 5. Creating Log File (with increasing LSN)

 ISSN: 2321-8169
3352 – 3359

In figure 5 we retrieve a global variable on the initiation of the main code that holds the filename of the log to be created (client). If this filename does not exists in the folder that holds the clients logs, the file is created and the event handler is pointed to that newly created log file.

B. WRITING TO RESPECTIVE LOGS (CLIENT LOGS/SERVER LOG VOLUME)

Providing the file path to the respective log file we can log changes made any transaction. This is done in our case by appending contents of an array, each line representing a new array index.

```
public static void WriteLinesToFileLog(string filePath, string[] lines)
{
    if (filePath == null || filePath.Length == 0)
        return;
    if (lines == null || lines.Length == 0)
        return;


    try
    {
        if (File.Exists(filePath))
            Home.fileWriter = File.AppendText(filePath);
        else
            Home.fileWriter = File.CreateText(filePath);

        foreach (string line in lines)
            Home.fileWriter.Write(line);
    }
    finally
    {

            Home.fileWriter.Close();
    }
}
```

Figure 6. Method to write changes made by transaction to log

The same module is applied to the single log volume at the server side. In this case logging made here represents the name of the log file the server has received in its circular buffer and the date and time it was received.

```
public static void WriteLinesToFileServer(string filePath, string[] lines)
{
    if (filePath == null || filePath.Length == 0)
        return;
    if (lines == null || lines.Length == 0)
        return;


    try
    {
        if (File.Exists(filePath))
            Home.fileWriter = File.AppendText(filePath);
        else
            Home.fileWriter = File.CreateText(filePath);

        foreach (string line in lines)
            Home.fileWriter.Write(line);
    }
    finally
    {
        Home.fileWriter.Close();

    }
}
```

Figure 7. Method to write changes made to the circular buffer

Figure 7 show a similar snippet with a different method name.

C. SENDING CLIENT LOGS TO SERVER

After the client has closed the write for a particular is sends it to the server on transaction commit. The server receives this log file into its circular buffer and then logs the of the log file sent by the client and the date and time the server received it. In our case a copy of the log file is sent to avoid re-logging at the server side. Once this activity has taken place it is now safe to flush changes permanently to storage. Figure 8 shows us a snippet on how logs are transferred from the client side of the architecture to the server side of the architecture.

```
try
{

    string sourceFile = System.IO.Path.Combine(@"C:\Users\
    Projects\ARIES (application)\ARIES (application)\Logs", ClientLog.globalClientLogToBeSentToserver);
    string  destFile = System.IO.Path.Combine(@"C:\Users\
    Projects\ARIES (application)\ARIES (application)\Logs\LogsFromClients", ClientLog.globalClientLogToBeSentToserver);

    System.IO.File.Copy(sourceFile, destFile, true);

    string[] serverlog = new string[10];
    int stringcount2 = 0;

    serverlog[stringcount2] = "The file '" + ClientLog.globalClientLogToBeSentToserver + "' has been successfully received from the client at " +
     DateTime.Now.ToLongTimeString();
    stringcount2++;

    Home.WriteLinesToFileServer(Home.filepathtoserverlog, serverlog);
}
catch { }
```

Figure 8. Transferring log file from client to server

XIV. OBSERVATION

A problem that arises due to the expense of communication between clients and the server is the inability of clients to efficiently assign log sequence numbers. The original ARIES algorithm requires that log sequence numbers are unique within a log, and that log records are appended to a log volume in a monotonically-increasing log sequence number order. In a centralized or shared memory system, this is easily achieved, since a single source for generating LSNs can be cheaply accessed each time a log record is generated. However, in a client-server environment, clients generate log records in parallel, making it difficult for them to efficiently assign unique LSNs that will arrive at the server in monotonically increasing

order. Furthermore, if the LSNs are to be physical (e.g., based on log record addresses), then the server would be required to be involved in the generation of LSNs.

To summarize, the issues that were addressed in a client-server environment were the following:

- The assignment of state identifiers (e.g., LSNs) to place on pages.
- The need to make undo a conditional operation.
- Changes to the Analysis pass of system restart to ensure correctness.

XV. Conclusion

In this thesis, we have described the problems that arise when implementing recovery in a client-server environment, and have presented an extended method that addresses these problems. The recovery method was designed with the goal of minimizing the impact of recovery-related overhead on networks during normal processing, while still providing reasonable rollback and system restart times based on the original ARIES algorithm. The method adopts efficient buffer management policies, allows flexibility in the interaction between clients and the server, and allows clients to off-load the server by performing much of the work involved in generating log records. Overhead will be reasonable. The study also raised issues to be addressed when applying this method in centralized databases rather than distributed databases. These issues include: reducing log record size, batching writes to the log disk, prefetching from the log during recovery, and exploiting additional parallelism between logging operations on the server and other operations on the client during normal processing.

XVI. Recommendation

Additional studies of realistic workloads of other architectures will be required in order to obtain a better understanding of the performance impact of the distribution of logging and recovery subsystems. Further research would have to be done to extend the recovery system to include media recovery, ARIES-RRH during the undo phase, and support for inter-transaction caching. Finally, this work has raised a number of interesting possibilities for applying alternative locking strategies to ARIES, and investigations should be carried out on the performance tradeoffs among these alternatives.

References

[1] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955.

[2] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", IBM Research Report RJ6649, IBM Almaden Research Center, November, 1990.

[3] Elmasri, R., & Navathe, S. B. (1994). Fundamentals of Database Systems 4th Edition. Pearson.

[4] Database Fundamentals - Data Security and Recovery http://www.personal.psu.edu/glh10/ist110/topic/topic07/topic07_08.html

[5] Coronel, Carlos, Peter Rob. Database Systems, sixth ed. Thomson Course Technology, 2004.

[6] Ricardo, Catherine. Databases Illuminated, second ed. p386-387 Jones & Bartlett Learning, 2012.

[7] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM Transactions on Database Systems, Vol. 17, No. 1, March 1992

[8] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. The Recovery Manager of the System R Database Manager, ACM Computing Surveys, Vol. 13, No. 2, June 1981.

[9] Crus, R. Data Recovery in IBM Database 2, IBM Systems Journal, Vol. 23, No. 2, 1984.

[10] Borr, A. Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-Processor Approach, Proc. 10th International Conference on Very Large Databases, Singapore, August 1984.

[11] Mohan, C., Narang, I., Palmer, J. A Case Study of Problems in Migrating to Distributed Computing: Page Recovery Using Multiple Logs in the Shared Disks Environment, IBM Research Report RJ7343, IBM Almaden Research Center, March 1990.

[12] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., "ARIES: A Transaction Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", IBM Research Report RJ6649, IBM Almaden Research Center, November, 1990, to appear in ACM Transactions on Database Systems.

[13] Kumar, V. Transaction Management Concurrency Control Mechanisms, 2012 http://sce.umkc.edu/~kumarv/cs470/transaction/T-management.pdf

[14] Haerder, T., Reuter, A., "Principles of Transaction Oriented Database Recovery - A Taxonomy", Computing Surveys, Vol. 15, No. 4, December, 1983.