Evaluating the Effectiveness of Fault Tolerance in Replicated Database Management Systems

Maitrayi Sabaratnam, Maitrayi.Sabaratnam@idi.ntnu.no
Norwegian University of Science & Technology(NTNU)

Øystein Torbjørnsen, ClustRa AS
Oystein.Torbjornsen@clustra.com

Svein-Olaf Hvasshovd, NTNU
Svein-Olaf.Hvasshovd@idi.ntnu.no

Abstract

Database management systems (DBMS) achieve high availability and fault tolerance usually by replication. However, fault tolerance does not come for free. Therefore, DBMSs serving critical applications with real time requirements must find a tradeoff between fault tolerance cost and performance. The purpose of this study is two-fold. It evaluates the effectiveness of DBMS fault tolerance in the presence of corruption in database buffer cache, which poses serious threat to the integrity requirement of the DBMSs.

The first experiment of this study evaluates the effectiveness of fault tolerance, and the fault impact on database integrity, performance, and availability on a replicated DBMS, ClustRa[6], in the presence of software faults that corrupt the volatile data buffer cache. The second experiment identify the weak data structure components in the data buffer cache that give fatal consequences when corrupted, and suggest the need for some form of guarding them individually or collectively.

1. Introduction

Increased global trade and travel demands that services provided by database applications, for example, in banking and telecommunication, are available 24x7x52 hours an year, in addition to the traditional requirement of preserving the integrity of the user data. In other words, the responsibility of masking both hardware and software failures, maintenance activities, and other natural or man-made catastrophes lies on the DBMS software.

These requirements are fulfilled to a great extent, by introducing fault tolerance. Fault tolerance can be achieved by redundancy, e.g., by a replicated DBMS. A replicated DBMS runs DBMS processes on different computers in a coordinated way. A computer running such a DBMS process is called a **node**. When a fault is detected in one node, the node is recovered. If this is not possible, the system will reconfigure such that the faulty node is excluded from the system configuration and its functionality is taken over by one of the replica nodes. Error detection, recovery, and reconfiguration must be done online automatically in order to achieve high availability .

Integrating fault tolerance in DBMSs as mentioned above need redundant hardware and enhanced software for effective fault detection, recovery, and reconfiguration. The enhanced software containing redundant execution of transactions and rigorous checks for error detection not only increases the software development cost, but also increases the DBMS's transaction response time. Therefore, the DBMS products serving critical applications with hard or soft real time requirements must find a tradeoff between fault tolerance and performance.

Though replication solves many problems regarding dependability, it is not a panacea. Replication enables controlled maintenance of hardware and software on a computer running a DBMS software, without compromising the DBMS availability. Further, replication limits the loss of data or DBMS unavailability due to some failures caused by operator errors or natural catastrophes. Still, it has some limitations. It is difficult to achieve complete error detection or fault-free software/hardware. Two situations that can give fatal consequences on availability and integrity are: 1) Simultaneous failure of all DBMS replicas, due to either fatal error propagation or accidental concurrent errors, making the system unavailable, 2) User data corruption (though the definition of fatal failure varies according to the requirements of the applications).

Software reliability has not kept pace with the hardware reliability over the past years. In the present systems, software is the bottleneck in achieving dependability, due to the complexity and the human factor involving in it. The software faults causing overlay errors are hard to trace and there impact are much higher than the regular errors [10].

The main goal of the DBMSs is to maintain the integrity

of the database image. DBMSs cache parts (or whole) of the database image in data buffer area before processing, and later flush the processed data to a persistent medium in order to preserve the durability of the data. Residual hardware or software faults may result in corrupting the buffer area. In addition, DBMSs have increasingly started to integrate special-purpose application code in order to give extensibility and flexibility to applications. This application code is allowed to access data buffer area. This makes the buffer area vulnerable to get corrupted by unintentional application codes. Among the possible faults occur in a DBMS, one of the most interesting and important class of faults is the one that corrupt the buffer cache, since it is the one that could give a serious blow to the integrity requirement.

Gaining an insight in 1) situations that give fatal consequences, 2) how can they be masked, and 3) what is the overhead for such masking is valuable in building fault tolerant DBMSs.

Even though many DBMSs being built on shared-nothing architecture include fault tolerance by replication in their systems, e.g., Tandem Remote Duplicate Database Facility, Sybase Replication Server, Oracle7 Symmetric Replication Facility , IBM Remote Site Recovery, and Informix Online Server, there are very few studies highlighting the above mentioned aspects.

Ng and Chen [9] integrated reliable memory for caching data into Postgress DBMS in three different ways and used fault injection to evaluate them. Chillarege [2] studied the failure characteristics of the commercial transaction processing system, IBM's IMS, Version 1.3. These studies were performed on centralized database systems.

Our study improves the state of the art in the following ways:

It extends the idea of evaluating fault impact further to the next generation DBMSs, which are distributed, and replicated in order to achieve fault tolerance and thus, having increased complexity. Introduction of fault tolerance requires that the evaluation of fault impact incorporates the issues related to fault tolerance, such as, failure masking by replicas, error propagation to replicas, and so on. In addition, we concentrate on the effect of faults on the integrity of user data.

Our study reconciles concepts from separate research communities, fault tolerant computing systems and database systems. It applies fault injection to evaluate different coverage and performance parameters in a replicated DBMS context.

It introduces a generic fault injection method that can be adopted by any DBMS from the design/prototype phase throughout the development and testing phase to evaluate the effectiveness of the fault tolerance mechanisms and the performance tradeoff. This early evaluation enables the development team to improve design decisions as well as to detect and correct design faults at an earlier stage. Implementation of this method needs little effort since it exploits the existing client interface of a DBMS. Fault injection takes a few instructions and thus, the intrusion is minimal. In this experiment, this method is applied to the data structure in the data buffer area. This method can also be applied to other DBMS specific structures like log buffer, locks, message, etc. Data buffers are chosen because of their crucial role in DBMS integrity as mentioned earlier.

Different error scenarios are created by injecting errors into a test DBMS, ClustRa - version 1.1. The impact of these errors is analyzed and the performance degradation, the time the system is in reduced fault tolerance level, and coverage statistics are collected. Section 2 describes the testbed where the experiment was conducted. Section 3 describes the experiment setup and process. Section 4 analyzes the results. Section 5 contains a summary, conclusion, and future work.

2. The Testbed

ClustRa Architecture

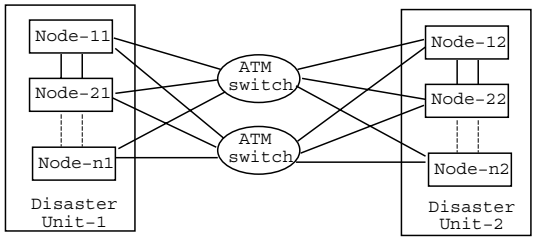

Figure 1. ClustRa architecture.

ClustRa provides a highly available, high performance, and fault tolerant DBMS platform for applications requiring soft real-time response time, e.g., applications in telecommunication. It is a replicated DBMS with a shared-nothing architecture, running on off-the-shelf UNIX work-stations. A **ClustRa node** is a work station running a ClustRa process. Nodes are grouped into different disaster units (having independent failure modes) connected by duplicated communication lines with high bandwidth. Duplication is used at process, data, and communication levels in order to minimize service unavailability and data loss, see Figure 1.

Data is partitioned into fragments and each fragment is replicated into **primary** and **hot standby** copies. They are placed in different nodes belonging to different disaster units, such that, the union of replicas in each disaster unit makes the whole database. Figure 2 shows how the data is partitioned and replicated among 4 nodes and 2 disaster units. Assume that the data is partitioned into 4 fragments, F_1, F_2, F_3, and F_4. Each fragment is replicated into two replicas, e.g., F_1 into R-111 and R-112 which are

placed in Node-11 and Node-12 belonging to DisasterUnit-1 and DisasterUnit-2 respectively, and so on. **Counterpart nodes** are those nodes which contain the copies of the same data. E.g., in Figure 2, Node-11's counterpart is Node-12 and Node-21's counterpart is Node-22. Failures are handled at node granularity, i.e., a failure causes the ClustRa process to restart at that node it resides.

Active nodes contain data while **spare** nodes do not. Spare nodes are on-line, can serve as coordinators for client transactions. In case of an active node failure, a spare node from the same disaster unit can take over the failed node's role, if the failed node does not get repaired within a specified time. This reduces the time window the system functions without replication.

Each node runs a ClustRa process and a supervisor process. The latter manages (starts, stops, etc) the ClustRa process as well as sending heart beats to its neighbor nodes. A node is pronounced as dead by its neighbor nodes, if it does not send a heart beat within a timeout period. This message is spread to all nodes and a virtual partition management protocol is used to maintain a consistent set of available nodes and services[6]. A primary-hot standby coordinator process pair coordinates a transaction among primary-hot standby participant process pairs, using a 2-phase commit protocol [4]. Coordinator and participant processes are executed as light-weight threads within the ClustRa process. Transactions are executed atomically in a **2-safe manner**, i.e., they are reflected in both the primary and hot standby before commit or not reflected at all.

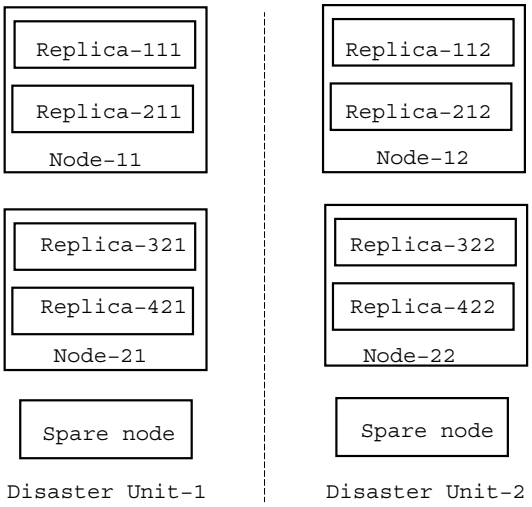

Figure 2. Data distribution in ClustRa.

ClustRa's Fault Tolerance Mechanisms: ClustRa masks all single node and single communication line failures. Different consistency checks are used to detect errors. Detected errors are categorized into three levels: normal, user, and serious. The first two error levels allow the system to continue functioning. User level errors do not harm the database integrity, therefore, the DBMS does not take any action other than informing the user about the error. Detection of serious errors makes ClustRa be fail silent [4].

A primary node failure is masked by its counterpart hot standby by **taking over** the primary role. The failure is detected by the missing heart-beat. During the takeover, some response to client transactions may time out for a fraction of a second, depending on the heart beat timeout. At the same time, the failed process is restarted by the supervisor process and an **online repair** is undertaken [1]. When the node (or eventually the spare) finishes repair, it **takes back** the primary role in order to balance the load.

3. Fault Injection Experiment

In complex systems like DBMSs, it is difficult to trace faults causing failures and system crashes. This is partly because the crash may leave incomplete or destroyed logs or the error has long latency. It is especially difficult in large, complex, parallel systems where the failure scenarios can not be reproducible since the order of the external events is not reproducible. Fault injection is used in order to identify and understand potential failures, their impact, and the system's ability to mask them, before the system is being operational, i.e., before the failure data from the field is available. Different hardware and software implemented fault injection (SWIFI) techniques are described in the literature. An overview of fault injection techniques and some tools is described in [5].

This section describes the experiment setup, underlying fault model used in the experiment, workload and fault injection process.

3.1. Testbed

A four-node ClustRa cluster is used in this experiment. Each disaster unit consists of one active and one spare node. A node is a Sun Ultra work station with a 200 MHz UltraSPARC processor, running ClustRa v.1.1, as illustrated in Figure 4.

3.2. Fault Model

Extensive field error reports and surveys are not available in young products like ClustRa, therefore we have to base our fault model on studies conducted in operating systems and databases [10, 11, 8]. Common software faults causing overlay errors are: assignment faults: e.g., code line *PreviousLog = tmp;* instead of *PreviousTrLog = tmp;* corrupts both PreviousLog and PreviousTrLog variables, initialization faults: wrong or forgotten initialization of vari-

ables, wild pointers: assignment or initialization faults to variables pointing memory locations, copy overflow: program copies bytes past end of buffer, type mismatch: a field is added to a structure, but all of the codes using that structure have not been modified to reflect the change, memory allocation: using a memory area after it has been freed, and undefined state: the system goes into a state unanticipated by the program may cause overlay errors since the invariant does not hold in this state.

The fault model that is relevant for our study is the software faults causing transient overlay errors that corrupt the data buffer area of the shared memory. Though overlay errors may corrupt any part of code or data, the ultimate (direct or propagated) corruption that affects the customers are those occur in database image. Since this corruption must be prevented in order to achieve the integrity goal of the DBMSs (see Sec. 1), the interest of this work lies in studying system behavior given that a corruption is occurred in the data buffer area. Since this corruption is a small subset of the errors resulting from the software faults causing overlays, we inject errors directly in the data buffer area in order to accelerate the database corruption, instead of injecting faults that result into direct or indirect corruption in data buffer area[1].

The question arises here is the validity of the mapping between the representative faults mentioned in the fault model and the errors being injected in this way. Christmansson and Chillarege [3] have proposed answers to the question of generating representative error sets that can be used in fault injection experiments, based on the field defect data for IBM OS. They address issues like what errors should be injected in which software module and when the injection should take place.

3.3. Workload Generator

The main purpose of the workload generator is to create an environment that can activate the error, i.e., it is accessed, propagated, or it crashes the system. The workload generator starts three parallel transaction clients (TC). Each TC sends single-tuple transactions to the DBMS, i.e., a transaction performs only one of the following operations on a single data record: insert, delete, update, or read. A TC is allowed to function 1) until it executes a maximum of 10,000 transactions sent in a back-to-back manner or 2) for a maximum of 300 seconds. 55% of the transactions are updates, 20% are read, and the rest is distributed among insert and delete such that at equilibrium, 75% of the inserted data exist. The size of the database is 4 MB at equilibrium. The data is divided into 3 tables, each is partitioned into 2 fragments, and each fragment is replicated. A data record consists of a primary key of type integer and a string type having variable length of maximum 512 bytes. Access to a data record is uniformly distributed, having no hot-spots. The workload is typical for telecommunication applications, for e.g., those setting up calls. Further, the transaction load is selected in order to exploit the system resources like CPU at the server to their maximum. The workload is designed to accelerate the access of a corrupted location by reducing the database size and increasing the number of transactions accessing the corrupted data.

[1] In [2], Chillarege inject overlay errors by overwriting a randomly chosen page of the real storage with hexadecimal 'FF's in order to accelerate an experiment studying the failure characteristics of the commercial transaction processing system, IBM's IMS, Version 1.3., instead of injecting faults.

3.4. Experiment Process

There are two experiments conducted. In the **first experiment**, errors are injected into a random area of the data buffer. This is referred to as error type *DBbuffer* in Sec.4. Start position for the corruption is uniformly distributed over the data buffer area. Number of bytes being corrupted is based on the study done by [10], but adjusted according to our software, platform, and experience. The distribution of number of bytes being corrupted is: 60% 1-4 bytes, 35% 5-1024 bytes, and 5% 1-9 KB.

This kind of general corruption does not give any specific information about 1) the data structure components lacking validity checks or 2) the causal relationship between the error source and the severity of the error impact, in other words, which particular component in the data buffer gives worst impact. If the answer to these questions are known, the weak component can be guarded with better detection techniques to limit the error impact. Therefore, we introduce specific errors in the **second experiment**. Here, we inject specific errors into particular components of the data structure covering the data buffer area.

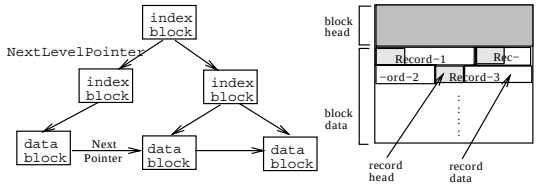

Figure 3. a) Structure of the data buffer area in use: index and data blocks are arranged in a tree structure. b) the layout of a block.

Data buffer area consists of used and unused data blocks. In order to accelerate the failures, we inject faults into the used area. Blocks in the used area are arranged in a B-tree structure, as shown in Figure 3. Each block has a header part and a data part. The data part consists of data records. The

header part of a block consists of the following administrative data: block identifier, number of records in the block , number of free bytes in the block, high water mark - the position where a new record will be inserted, and a pointer to its next block (found in data blocks only). A record also has an administrative part and data part. Administrative part contains: a key descriptor describing the number of fields used to identify a record uniquely and an administrative descriptor stating whether a record has the knowledge about the log record that contains information about the last change a transaction made on the record. Data part of an index record contains access path to a block in next level. Data part of a data or leaf block contains user data. In the following sections, errors overlaid on these components are referred to as error types: *BlockId, NoOfRecords, NoOfBytesFree, HighWater, NextPointer, KeyDescriptor, AdmDescriptor, NextLevelPointer,* and *UserData*.

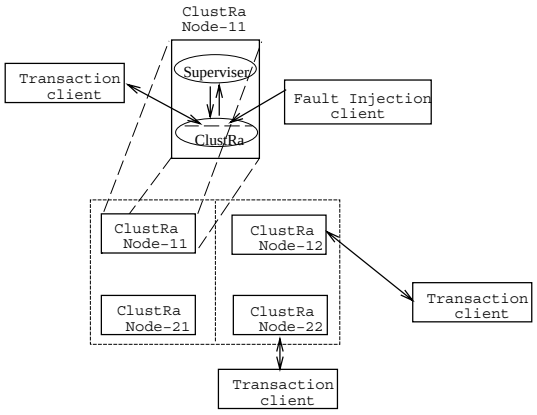

Figure 4. The experiment setup (ClustRa Node-11 is zoomed). Initially, Node-11 and Node-12 are active, and Node-21 and Node-22 are spare.

An experiment run consists of starting the DBMS and the workload generator, injecting an error, stopping the DBMS, and analyzing the logs, and is conducted by an experiment manager (EM). ClustRa processes are started on four ClustRa nodes as illustrated in Figure 4, two spare nodes and two active nodes. Active node-11 has primary data and node-12 has hot standby data. The workload generator is started 240 seconds after the DBMS is started, it is allowed to stabilize for 30 seconds, and then the fault injector is called with one of the error types mentioned above with relevant parameters. The workload is maintained for nearly 300 seconds. Then, the DBMS is requested to give the contents of the database. The DBMS is stopped after 300 seconds. In the analysis phase, the database content is compared with the database content maintained by the TC in order to see any discrepancy. Besides, each client checks each response delivered by the DBMS against its internal database for each transaction. The DBMS log containing repair information is also analyzed and the repair time is calculated.

There were 725 runs conducted in the first experiment, 50 runs for each specific error type in the second experiment, and 50 error-free or golden runs, totaling 1225 runs. The clock time used by each run is around fifteen minutes, giving a total of 306 hours. In order to create different fault scenarios as possible, each run was started with a different seed such that the choice of a block or position to be injected and the injection point in time varied. Therefore, the experiment did not evaluate the repeatability of the error impact.

The measures from the golden runs are used to evaluate the effects of fault tolerance mechanisms on performance in the presence of errors. In order to make the golden runs comparable to runs with fault injection, a golden run also executed the fault injection code, but overlaid an unused location instead.

3.5. Fault Injector

The fault injector (FI) injects error types mentioned in Section 3.4 into the data buffer area. The fault injector client (FIC) can also be started on a non-ClustRa node like a TC. FIC is started at a point in time distributed uniformly between 30-60 seconds after the workload generator is started. FIC is given the error type and the relevant parameters by the EM. For example, if the error type is BlockId, then the parameters will be a seed value used by EM to repeat the run if necessary, a data table identifier, and a flag saying whether the chosen buffer block is an index block, a data block, or any of them. FIC then sends a message containing the fault injection request together with the parameters. The DBMS server interface is extended to handle requests from a FIC. At the receipt of this message, this request is handled like the requests from the TCs. To process this request a small piece of software code is added to the DBMS server to access a memory address or data buffer component as specified by the parameters and overlay a random bit pattern accordingly. This takes very few machine instructions. In addition, comes one extra message-receiving cost per run. In order to enhance the analysis of the fault impact, ClustRa is run on trace-mode, i.e., it wrote information, such as, node crash detection time and the outcome of restart (success or not), and the restart finishing time to a trace file. This may result in some timing difference in the execution.

4. Analysis of Results

This section evaluates the fault tolerance of ClustRa from the experiment results. Two aspects of fault tolerance are

Error Type	# errors detected	# failures masked
Exp-1: DBbuffer	442(61%)	431 (97.5%)
Exp-2: NextLevelPointer	50	48
UserData	0	-
KeyDescriptor	48	47
AdmDescriptor	16	16
BlockId	50	50
NextPointer	25	24
NoOfRecords	50	49
NoOfBytesFree	1	1
HighWater	39	39
Total	279(62%)	274 (98%)

Table 1. Error coverage.

Error Type	# double failure	# data corruption
Exp-1: DBbuffer	11(1.5%)	11(1.5%)
Exp-2: NextLevelPointer	2	0
UserData	0	50
KeyDescriptor	1	0
AdmDescriptor	0	1
BlockId	0	0
NextPointer	1	0
NoOfRecords	1	1*
NoOfBytesFree	0	0
HighWater	0	7
Total	5(1%)	59(13%)

Table 2. Fatal failures. Data marked by '*' is explained in Fatal Failures section.

evaluated: effectiveness of error detection and efficiency of recovery.

4.1. Effectiveness of Error Detection

Effectiveness of error detection is measured by error coverage and fatal failures.

Error Coverage: Table 1 shows the coverage statistics. The first column shows the number of errors that are detected by the system and the error detection coverage - detected errors as a percentage of the total injected errors. Detection of an error causes the node to crash and a node failure is masked by a hot standby takeover. The second column shows the number of successful masking and coverage. In the first experiment, 61% of the 725 injected errors were detected. 97.5% of the detected failures were masked, except for the **double failures**, where both active nodes crashed almost simultaneously within a very short interval (see Table 2 for double failure statistics). The spare nodes cannot take over in this case because there is no available active node to copy data from online. In the second experiment, the corresponding figures were: 62% and 98%.

Locations *UserData* and *NoOfBytesFree* show very low detection, 0 and 1 cases respectively. Low error detection means that the erroneous locations were either not accessed, overwritten, or read without being tested. The experiment did not have enough instrumentation to verify whether an erroneous location is accessed or not, except from the failure manifestations and thus, cannot differentiate the errors overwritten and the dormant ones. However, the observed fatal failures of Table 2 shows clearly that the *UserData* is read without tested and contribute to severe integrity loss.

Fatal Failures: Double failures and user data corruption are defined as fatal failures from the severity of their consequences. The former makes the DBMS unavailable until the data is restored from a backup and also causes the recently committed transactions to be lost. The latter affects the integrity of DBMSs. Table 2 shows that there were 11 double failures in the first experiment (1.5%) and 5 double failures (1%) in the second one. Five of these 16 cases were traced to two design faults in the recovery mechanism, which were accidentally provoked on the recovering node and its counterpart almost simultaneously. The rest might be resulted from error propagation, but could not be verified.

User data corruption is detected by the client by comparing the DBMS reply with its internal database. In the first experiment, 11 runs shows user data corruption. In the second experiment, the corresponding figure was 59. In the second experiment, the majority (50) comes from the *UserData* corruption since *UserData* have no detection mechanism associated with it. Apart from that, there were only 9 runs found with data corruption for the rest of 400 runs (2%) in the second experiment. This figure corresponds to that of the first experiment (1.5%), where the choice of *UserData* for corruption was random, not as deterministic as 50 out 450 of the second experiment.

HighWater is very much vulnerable to data corruption (7 out of 50 runs). In 4 of the 9 runs, one or more committed transactions were missing. This can be the consequence of the design fault found in the double failure case or another design fault found in the checkpoint mechanism. One of these 4 cases show severe inconsistency in user data (marked by '*'). Further analysis shows that this case had consecutive node failures, i.e., after the first node crashed and repaired itself, the counterpart node crashed and re-

Error Type	Repair Time (sec)	Online repair	Rep.not finished
DBbuffer	124	315	116
NextLevelPointer	160	22	26
UserData	0	0	0
KeyDescriptor	147	42	5
AdmDescriptor	124	10	6
BlockId	110	40	10
NextPointer	143	19	5
NoOfRecords	134	43	6
NoOfBytesFree	66	1	0
HighWater	155	30	9

Table 3. Time taken to restore the replication level, number of cases where the online repair succeeded and not succeeded.

Error Type	**Performance Degradation**
DBbuffer	0.121019
NextLevelPointer	0.146497
UserData	0.0127389
KeyDescriptor	0.210191
AdmDescriptor	0.0764331
BlockId	0.146497
NextPointer	0.0764331
NoOfRecords	0.171975
NoOfBytesFree	0
HighWater	0.191083

Table 4. Performance degradation due to the errors.

paired itself, before the occurrence of double failure. Since this was the only case where consecutive node crash was found, it is difficult to draw any conclusions, but it may indicate an error propagation or some other design fault in the recovery mechanism. Due to the non-repeatable nature of the experiment, the fault could not be traced.

4.2. Efficiency of Recovery

Efficiency of recovery is characterized by the period of time the system is in reduced fault tolerance level and the performance degradation due to fault tolerance activities.

The period of time system is in reduced fault tolerance level: This is the time the system runs without replication after a node crash. It is calculated as the time between a node crash and a successful repair of it (or the spare). This period is very crucial because if the functioning replica node fails before the repair is finished, a double failure will occur. The longer the system is in a reduced fault tolerance level, the higher the danger to get a double failure. Column 1 of Table 3 shows the period of time the system was in a reduced fault tolerance level. *NextLevelPointer, HighWater, KeyDescriptor, NextPointer, NoOfRecords*, and *AdmDescriptor* show long repair time, and the double failures (Table 2) have occurred within these error types. The exceptions *HighWater* and *AdmDescriptor* indicate the nondeterministic nature of events causing double failure, depending on the transaction pattern and load, error propagation to the counterpart node, and the error detection mechanisms.

Columns 2 and 3 of Table 3 shows the node failure cases where the online repair was succeeded and not. The error type *NextLevelPointer* shows severe problems, having half of its runs unable to finish the repair. Further tracing shows that 80% of the repair-not-finished cases in the first experiment and 96% in the *NextLevelPointer* indicated a design fault in the repair mechanism. Due to this fault, the crashed node and the spare got into a deadlock situation such that none of them were able to complete the repair.

In two of the runs belonging to *NextPointer* error type, the failure was masked by recovery even though the error was not really removed. This dormant error had the potential to crash the process when being accessed next time. One of the runs showed a design fault in the repair algorithm, where the crashed and the spare node from the same disaster unit *both* repaired themselves successfully, which should not occur according to the replication semantics.

Performance degradation reflects the fault tolerance overhead. Performance degradation is measured from the clients' point of view. Each run measures the number of transactions succeeded per second (TPS). Average TPS for faulty runs is calculated for each error type. This average is compared with the average TPS of the 50 golden runs.

$$PerformanceDegradation_{ErrorType_i} =$$

$$1 - \frac{AverageTPS_{ErrorType_i}}{AverageTPS_{golden}}$$

The results are presented in Table 4. The repair activity generally has an adverse impact on performance. After a node crash, the active counterpart node must help the crashed node to recover in order to reestablish the fault tolerance level. In addition to serving the usual TCs, the active node must send the database image if necessary and the changes occurred in the database after the node crash, to the recovering node. This extra activity reduces the TC throughput. As shown in Table 4, this is evident in *UserData*, *NoOfBytesFree*, *AdmDescriptor*, and *NextPointer* which have less error detection coverage, and hence, less repair activity, and as a result, less throughput loss.

On the contrary, *KeyDescriptor, HighWater, NoOfRecords, NextLevelPointer*, and *BlockId* have high detection coverage and repair activity and show high performance degradation. Variations like *BlockId* having 50 error detection has 14% throughput loss while *KeyDescriptor* having 48 error detection has 21% throughput loss could have been caused by the factors such as, the size of the database and the period of time the crashed node was away at the time of the repair, which are not tracked in the experiment.

In order to get a real life statistics, the performance should be corrected to undermine the cost of fault injection and executing the DBMS in the trace-mode where it writes trace information regarding repair to a log file.

5. Summary, Conclusion, and Future Work

Fault tolerance ability of a replicated DBMS and fault impact are evaluated in the presence of software errors corrupting data buffer area in the DBMS process's shared memory. A generic fault injection method, extending the existing client interface, is used to inject errors in different data structures of a DBMS. Data buffer area is chosen in this experiment due to its crucial role in a DBMS.

Error detection and failure masking, together with the impact of faults on integrity, performance, and availability are evaluated. General corruption of the first experiment shows that 61% of the 725 errors are detected. All detected errors caused in node failures. All the node failures are masked by hot standby replica node taking over, except in the 1.5% double failure cases. The fault tolerance level is restored successfully in 71% of the failure cases, but in 26% of the failure cases, the repair was not finished. 91% of these repair-not-finished cases are caused by a design fault in the repair mechanism that put the repair in a deadlock situation. In 1.5% of the total runs show user data corruption. Two design faults in repair mechanism cause 2 double failures. A design fault in checkpoint mechanism and the one in repair mechanism (mentioned above) caused integrity problems. (Further, the study also identified another erroneous behavior of the system regarding to repair.) Performance degradation due to fault tolerance activities is 12%.

General corruption did not give enough feedback about 1) the weak components in the system, which have to be guarded in order to improve the fault detection and hence fault tolerance and 2) the causal relationship between the error source and the severity of the fault impact. Therefore, a second experiment was conducted by corrupting specific data components present in the data buffer area. We identified components which are very sensitive and affect database integrity (*UserData, HighWater*), availability (*NextLevelPointer*), and performance degradation (*KeyDescriptor*).

The results from the first experiment can be used to get an insight on fault tolerant DBMSs in general, while the results from the second experiment is more relevant to those DBMSs which arrange the data in a B-tree structure, which is the case with most of the traditional DBMSs. Further, fault injection proved to be an efficient complementary validation method. It provoked failure scenarios which other traditional testing methods did not do. In our case, this lead to trace 4 residual design faults in the recovery and checkpoint mechanisms.

The findings suggest that error detection mechanism connected to data buffer components needs an enhancement, such as, a checksum or some kind of memory scrubbing in order to guard the vulnerable components individually, or collectively at block level, or block head level together with record level.

A study similar to this is essential to evaluate the improvement on fault tolerance and tradeoff on performance by these proposed error detection methods. It would also be interesting to evaluate the dependability growth after the disclosed design faults being corrected. Further, Using standard workloads like TPC-C [7] will make the results of the study more interesting for a wider user community.

References

[1] S. Bratsberg, Ø. Grøvlen, S. Hvasshovd, B. Munch, and Ø. Torbjørnsen. Providing a Highly Available Database by Replication and Online Self-Repair. *International Journal of Engineering Intelligent Systems*, 4(3):131–139, 1996.

[2] R. Chillarege and N. Bowen. Understanding Large System Failures - A Fault Injection Experiment. *Proc. 19th. Ann. Int'l Symp. Fault Tolerant Computing*, pages 356–363, 1989.

[3] J. Christmansson and R. Chillarege. Generation of an error set that emulates software faults based on field data. In *Proc. 26th. Ann. Int'l Symp. on Fault Tolerant Computing*, 1996.

[4] J. Gray and A. Reuter. *"Transaction Processing: Concepts and Techniques"*. Morgan Kaufmann Publishers Inc., 1993.

[5] M.-C. Hsueh, T. Tsai, and R. Iyer. Fault Injection Techniques and Tools. *Computer*, April 1997.

[6] S. Hvasshovd, Ø. Torbjørnsen, S. E. Bratsberg, and P. Holager. The ClustRa Telecom Database: High Availability, High Throughput and Real Time Response. In *Proceedings of the 21st VLDB Conference, Zürich*, 1995.

[7] J.Gray, editor. *"The Benchmark Handbook for Database and Transaction Processing Systems"*. Morgan Kaufmann Publishers Inc., 1991.

[8] I. Lee and R. Iyer. Faults, Symtoms, and Software Fault Tolerance in the Tandem GUARDIAN Operating System. *Proc. FTCS-23*, pages 20–29, 1993.

[9] W. T. Ng and P. M. Chen. Integrating Reliable Memory in Databases. *Proc. of the 23rd VLDB Conference, 1997*, 1997.

[10] M. Sullivan and R. Chillarege. Software Defects and their Impact on System Availability- A Study of Field Failures in Operating Systems. *Proc. FTCS-21*, pages 2–9, 1991.

[11] M. Sullivan and R. Chillarege. A Comparison of Software Defects in Database Management Systems and Operating Systems. *Proc. FTCS-22*, pages 475–484, 1992.