An Efficient Scheme for Providing High Availability

Anupam Bhide Ambuj Goyal Hui-I Hsiao Anant Jhingran

IBM TJ Watson Research Center
Yorktown Heights, NY 10598

Abstract

Replication at the partition level is a promising approach for increasing availability in a Shared Nothing architecture. We propose an algorithm for maintaining replicas with little overhead during normal failure-free processing. Our mechanism updates the secondary replica in an asynchronous manner: entire dirty pages are sent to the secondary at some time before they are discarded from primary's buffer. A log server node (hardened against failures) maintains the log for each node. If a primary node fails, the secondary fetches the log from the log server, applies it to its replica, and brings itself to the primary's last transaction-consistent state. We study the performance of various policies for sending pages to secondary and the corresponding trade-offs between recovery time and overhead during failure-free processing.

1 Introduction

One promising approach to scaling up the performance of transaction processing architectures is to partition a database over loosely coupled multiple processors typically connected by a local area network and each having its own private disks and its own private memory. This is the **Shared Nothing** [13] approach, typified by commercial systems like Teradata's DBC/1012 [16] and Tandem's NonStop SQL [15]; and research prototypes GAMMA [4] and BUBBA [3]. In this scheme, a relation is divided into partitions which are then distributed over the multiple processors so as to balance loads. One problem with this approach is that the probability of all processors remaining up in an interval decreases exponentially in the number of processors[1]. Since many of the relations may be partitioned across all nodes, the failure of a single processor might prevent a large part of the workload from being executed. Thus, providing high availability in this environment is an important problem.

In this paper, we present a new mechanism for managing replicated data in a Shared Nothing environment. For simplicity of exposition, in this paper we assume that there are at most two copies of a particular database partition (in most cases the mean time to failure (**MTTF**) provided by two copies will be large enough). Our scheme can be easily extended to more than two copies. The two copies are commonly termed **primary** (against which database requests are directed), and **secondary** (which is used to take over the role of the primary, should the primary go down). Our *mechanism* for keeping the secondary replica up-to-date consists of asynchronously sending updated page images at appropriate points in time from primary to the secondary. We will study various *policies* which send pages at different times from primary to secondary and the resulting trade-offs between overhead and recovery time.

For concreteness of exposition, we assume that our algorithm will use the **ARIES** [10] recovery scheme to bring the database to the latest transaction-consistent state after a processor/disk failure; however our algorithm should extend to any log-based recovery scheme. Furthermore, we also assume that log records generated at each transaction processing node are sent to a log server node[2] for storage and log records produced by the primary can be accessed by the corresponding secondary. We assume that the log server can be made highly available using either replication or two servers sharing dual-ported disks. We ignore these issues in this paper. The log represents the disk states of both the primary and the secondary replicas in a unified manner. If the primary goes down, the secondary accesses the log server, obtains the log records and uses them to bring itself to the latest *transaction-consistent* state. This is the database state which has all the updates of all committed transactions and none of the updates of any of the transactions that were in-flight when the primary crashed.

Our design has a number of goals:

1. High availability should be provided with minimal sacrifice of performance in the failure-free case and without excessive resource overheads.
2. Special-purpose hardware should be avoided wherever possible because 1) they tend to be expensive, and 2) increase the probability of operator errors.
3. Irrespective of the sequence and number of failure and recovery events of the primary and secondary nodes for a given partition, there should be no data loss or inconsistency (no loss of updates of committed transactions and complete backing out of updates of uncommitted transactions).

[1] If the probability that a processor fails in an interval is p, then the probability that at least one processor in an n processor shared-nothing system is down is given by $1 - (1 - p)^n$.

[2] The log server can reside on one of the transaction processing nodes, or it can be on a special node that provides only logging service.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0236...$1.50

We can classify previous work on maintenance of the secondary copy in the following manner:

1. **Synchronous**: In this method, changes made to the database state at the primary are immediately reflected at the secondary copy. There are two distinct approaches in this form of replication:

   (a) **Physical**: In this, the bits on the primary's disk are copied, and there are no semantics associated with the bits. For example, Tandem Non-Stop SQL [15] achieves this by using dual-ported mirrored disks to achieve high availability, but requires special purpose hardware that violates our second requirement.

   Various extensions to RAID [12] (e.g., RADD [14] and Parity Striping [6]), in effect, provide a secondary without doubling the disk space overhead. However, RADD is effective mainly for disk failures and disaster recovery. Some auxiliary mechanism is require to handle the case where the secondary processor (in cases where group size is one) goes down for a short while, and then must play catch-up with the primary processor[3]. In addition, for group sizes greater than one, RADD (like RAID) suffers from excessive I/O overhead, since every write involves a read, modify and a write of a check block.

   (b) **Logical**: In this, the database state is replicated, by running the transaction on the two copies, and is exemplified by the Teradata DBC/1012 database machine [16]. However, this approach violates our first design goal because resource consumption (both CPU and I/O overhead) are significantly worse than the single copy case.

   Thus, for various reasons, synchronous updates are not the right mechanism in our environment.

2. **Asynchronous**: The transactions complete at the primary without the secondary reflecting the latest state. Most of the proposals using this mechanism employ spooling of the log to the secondary before committing a transaction [9, 11]. The log is stored at the secondary, is sorted and compressed and is then applied to the secondary replica. Research in this area is typically for disaster recovery. Consequently, their design considerations are different from ours. For example, communication is much more expensive in their environment. Our method trades I/O overhead and CPU overhead at secondary for network communication in comparison to the log spooling methods. Note that network communication is expected to become cheaper at a much faster rate than I/O and CPU. Also, since disasters are rare and usually result in the permanent loss of a node, these schemes ignore issues such as automatically switching operation back to the designated primary site.

This paper presents an asynchronous replica maintenance algorithm and performance analysis to explore the trade-offs between recovery time and run-time overhead. In order to analyze these trade-offs, we develop expressions for I/O overhead and recovery time for log-based transaction systems which apply to single-site database recovery as well as for databases having multiple replicas. Our model assumes that dirty pages are written to disk by a daemon which wakes up periodically, so as to bound recovery time. We assume that the main memory available for the database buffer is large enough and that recovery bounds are tight enough so that LRU mechanism is never used to write out a dirty page. A detailed single-site recovery analysis based on strict LRU is presented in [8].

The rest of the paper is organized as follows. In section 2, we describe in detail our replication algorithm, including the design of a unified log which keeps track the database states of both the Primary and the Secondary, Also discussed are the role of the node manager and both normal and failure-processing. In section 3, we present the various policies that can be used to implement our algorithm. The next section presents the performance analysis of recovery processing as a function of the I/O overhead we are willing to tolerate during normal processing. Finally, the paper ends with some conclusions in section 5.

[3]With a group size of one, RADD is in effect equivalent to a strategy where a disk write on primary does not return before its copy (either through the entire page image, or through a set of "xor" bits) is written out at the secondary.

2 The Asynchronous Replica Maintenance Scheme

The environment we consider is a Shared Nothing database machine architecture with a common log server. Having a common and highly available log server simplifies our design considerably; however, another good reason for assuming a common log server is that the cost of the log disks may be amortized over a number of nodes. The log server must be made reliable, perhaps by one of the techniques suggested in the introduction, but we ignore this aspect in the current paper.

We assume that there are at most two copies (also termed replicas) of a particular database partition (though our scheme can be easily extended to more than two copies). The two copies are commonly termed **primary (P)** (against which database requests are directed), and **secondary (S)** (which is used to take over the role of the primary, should the primary go down). Note that each node, will in general play the role of primary for one or more partitions **and** the role of secondary for one or more **different** partitions. Our algorithm will be described for one such partition, and must be repeated for each database partition, both during normal operations, as well as during recovery.

Our *mechanism* for keeping the secondary replica up-to-date consists of asynchronously sending updated page images at appropriate points in time from primary to the secondary. All log records are written to the log server by the primary; however, the log represents the disk state of both primary and secondary replicas. Thus, the secondary can recover from the log if the primary fails.

To completely describe the algorithm, we need to discuss the following:

1. the changes to the ARIES log needed to keep track of both replicas.
2. the actions taken by primary during failure-free operation to keep the secondary copy up-to-date.
3. system and node management actions, performed by a node manager, which involve detection of failure, assignment of primary and secondary roles and switching

of these roles. The node manager must take consistent actions in the face of lost messages, network partitions and other errors.

4. how recovery is achieved in a number of failure scenarios.

Before describing these functions in detail, we briefly present the salient features of ARIES relevant to our algorithm. The reader is referred to [10] for other details.

2.1 ARIES

ARIES maintains a dirty page table (**DPT**) for all pages that have not yet been pushed to the disk. This DPT contains two fields for each page that is dirty and in-memory: the **page-id**, and **RecLSN** (or recovery log sequence number, which is the address of the next log record to be written when the page was first modified). In other words, whenever a non-dirty page is updated in memory, a row is added to the DPT with the corresponding page-id and the next LSN to be assigned. Thus, the RecLSN indicates a position in the log from where we have to start examining the log to discover the log record that describes the changes made to this page. Whenever a page is written to the disk, the corresponding entry in DPT is deleted. Furthermore, at a checkpoint (**CP**), the current DPT is written to the CP log record.

When the database system is recovering from a crash, it goes through the log from the last checkpoint record (which contains the DPT at the time of the checkpoint) to the end, and reconstructs the DPT. In addition, it determines winner and in-flight transactions. This scan is called *analysis phase*.

The ARIES algorithm then goes through the log in the forward direction (starting at a position which is called the Minimum RecLSN, which is the minimum RecLSN of the entries in the reconstructed DPT, and indicates the earliest change in the database log that was potentially not written to the disk), examining all redoable actions. For each redoable action for a page that was potentially dirty at the time of the crash and which passes certain criteria, ARIES redoes the action.

Finally, in the third pass, it proceeds backward from the end of log, undoing all in-flight transactions.

We can now describe the four aspects of our algorithm mentioned above (namely, changes to ARIES log, normal operations, node manager, and recovery). For the purpose of the exposition, we will focus on a particular database partition that has two replicas: R_1 and R_2. One of these plays the role of the primary (P) and the other the role of the secondary (S) for this partition.

2.2 Design of the Unified Log

ARIES records information about which pages are in volatile storage (main memory) only in its checkpoint log records. All other log records are independent of the main memory state (or disk state). Since the disk states of the two replicas R_1 and R_2 will in general be different, the ARIES checkpoint must be modified to reflect this. All other log records will apply equally well to both R_1 and R_2.

P maintains two DPTs - DPT_{R1} and DPT_{R2}. Whenever a clean page is pinned in memory of P for updating, an entry is added to both the DPTs with current RecLSN value. When a page is to be forced to P's disk, it is simultaneously spooled to S's disk if it has not yet been spooled since the last write (we later describe various policies wherein such spooling might occur earlier than the write on P's disk). Whenever the disk write completes at R_i ($i = 1, 2$) and P receives the acknowledgment of the same, the corresponding entry in DPT_{Ri} can be deleted.

After a disk write for a page is started at P and the page is shipped to S, the page might need to be updated again at P. This results in two entries for the same page in the DPT of each replica for which the disk write is not yet complete. This is more likely for S since there is a round-trip message delay in addition to the disk write latency. As a consequence, the acknowledgments from S (and local disk writes) should not only reflect the PageID, but also the RecLSN number of the received page, so that P can delete the appropriate entry in DPT_{Ri}. Finally, the recovery algorithm must use the entry with the smallest RecLSN for recovery purposes.

If a current S is down, in effect, acknowledgements from S for disk writes are never received. Thus, if one were prepared to write increasingly larger DPT_{Rk} (if R_k is playing the role of S), then no changes need to be made to the algorithm. Let us define **SDownCP** to be the CP immediately preceding the time when S went down. Then, an effective way to bound the size of the DPT for the secondary is for P to write a a pointer to SDownCP in place of the DPT for the secondary in all subsequent CP's. In effect, P behaves as if it is the only replica, and hence writes only its DPT. If S must recover after being down, it must start the analysis phase from SDownCP[4].

2.3 Failure-Free Operation

Database operations are executed only on the primary replica. In order to keep replicas P and S reasonably synchronized with respect to the database state, updated pages are sent to S at some time before they are discarded from P's buffer. There are a number of policies possible for doing this. They involve different trade-offs between recovery time and CPU, disk and network overheads during failure-free processing depending on how soon after update pages are sent to S. We will study these policies and trade-offs in section 3. The only criterion we need for correctness is that the Write-Ahead Log protocol be used i.e. an updated page be sent to S only after the log record describing the update is written to the log. The set of updated pages is termed SDP, for Stream of Dirty Pages. Furthermore, S acknowledges to P when it writes dirty pages to its disk (it may buffer pages in its own memory for faster recovery).

Meanwhile, all log records are written to a log server. The log represents the disk state of both the primary and the secondary in a unified fashion, enabling either replica to recover to the latest transaction consistent state when required.

The primary carries out the algorithm described in Table 1 to update the secondary replica.

2.4 The Node Manager

We need a mechanism to ensure that inconsistent actions do not take place either because of network partitions, lost messages or other errors. For example, both the nodes which have copies of a given partition should not decide to take

[4]If desired, a CP can be taken immediately when S goes down, and this CP will become SDownCP.

```
for each dirty page in buffer
{
   if (S is up)
   {
      Send page to S after latest log record
      modifying the page is written out and
      before the page is expelled from the buffer.

      After receiving acknowledgement from S that
      page is on disk delete entry for page from
      DPT for S.
   }
   else
       /* Do nothing, i.e. P behaves as if it
          is the only replica                  */
 }

At checkpoint time do:
   if (S is up)
   {
       write DPT of both P and S in checkpoint.

       inform S that a checkpoint has taken place.
       /* For S to reset the ''Received List''
          described later                      */
   }
    else
       write DPT of P and a pointer to the latest
       SDownCP.
```

Table 1: Pseudo-Code for Normal Operations at Primary

```
Request last checkpoint from log server.
if (checkpoint has DPT for R1)
{
    Start ARIES analysis pass from this DPT.

    Delete pages from this DPT which
    occur in ''Received List''.
 }
 else
 {
     Follow pointer to SDownCP.

     Start ARIES analysis pass from this
     point and reconstruct (conservative)
     DPT before failure.
 }
  Perform REDO and UNDO ARIES passes and
  complete recovery.
```

Table 2: Pseudo-code for Recovery Process

over the primary role. We call this the *node manager* function. This mechanism is best implemented at the log server node since it can enforce its view of the system state by not allowing the "wrong" node to write log records. A node which cannot write log records cannot commit any transaction and hence can do no harm. In our scheme, the node manager keeps track of the state of each partition. If a primary fails, it asks the secondary to take over the role of primary after recovering its database state. If a secondary fails, it asks the primary to record SDownCP.

The complete design of the node manager is not discussed here because of lack of space; see [1] for details. Briefly, the node manager is informed by every node when it recovers after a failure. Based on a state table that the node manager keeps for every partition, it sends this node a message asking it to recover. The message also specifies the role it will play (primary or secondary) after it recovers.

2.5 Recovery

A node (say R_1) is asked to recover in one of the following two scenarios:

1. It comes back up after a failure: It first informs the node manager which then decides the role assignment for R_1. If it happens to be the secondary, then the node manager asks the corresponding primary to log an **SUp** record. The node manager then asks R_1 to recover, either till SUp, or till the end of the log.

2. It is asked to take over the role of the primary, in which case, the node manager asks it to recover till the end of the log.

The actions taken by R_1 during recovery are described in Table 2, and should be mostly self-explanatory.

The "Received List" (termed RL) is a list of pages S receives from the primary that are potentially in the DPT for S. The RL consists of the following: <recLSN, pageLSN, page> indicating the recLSN and pageLSN of the page received. For the purpose of recovery, the RL is null for P. Let us also define maxLSN(page) to be the maximum of pageLSN for the page in RL.

At checkpoint time, primary sends S's DPT over to S. S then deletes all entries from its RL for the pages that *do not appear* in this DPT. Remember that S is up when it is trying to recover the state of P, and hence its buffer state is not lost; consequently RL entries can be deleted for recovery purposes.

The following steps are taken by the recovering node to update DPT during the analysis phase:

1. For each <recLSN, page> entry in the DPT at the checkpoint, delete the entry if maxLSN(page) >= recLSN

2. For each log record <LSN, page> encountered subsequent to the checkpoint:

```
if (maxLSN(page) < LSN)
/* have not seen changes from this log */
   if no entry for page in DPT
       add an entry <LSN, page>
   else
       do nothing
```

3 Policies To Speed Up Secondary Recovery

We will now discuss policies for sending pages from primary to secondary and the trade-offs involved between recovery time and performance overhead. There are two techniques that could be used to speed up recovery at a secondary after a primary fails.

- Suppose that the buffer manager has a policy that it writes updated pages to disk if they are older than a certain threshold to bound recovery time. Then one reasonable policy would be to send a page over to the secondary when it is written to P's disk. To further speed up recovery at secondary, pages could be sent over to the secondary before they are written to primary's disk, specially if network bandwidth is cheaper than disk bandwidth. However, the WAL protocol must be followed and log records for this page must be written before an updated page is sent to the secondary. One policy would be to send updated pages to the secondary after the updating transaction commits on the primary (thus ensuring that log records are written).
- When a primary fails, and a secondary is recovering, it must read pages in the DPT from disk to apply log records to it during the redo and undo phases of ARIES. The disk I/O would not have to be performed if the secondary buffered hot[5] and often updated pages in main memory. The secondary can get hints from the primary about which pages were good candidates for such buffering. The primary could keep information about how long each page has been in its buffer and thus provide such hints.

Based on the above discussion, we will study the recovery time and overheads for the following three policies.

1. **Simple**: In this policy, a dirty page is sent to the secondary whenever it is written to the disk at the primary. The secondary attempts no buffering and writes these pages out as they come in.
2. **Secondary Buffering**: This policy attempts to buffer hot pages at the secondary to save I/O. During recovery, only cold pages need to be read and this reduces recovery time.
3. **Commit and Send**: In this policy, an updated page image is sent to the secondary as soon as the corresponding transactions commit on the primary node and all log records associated with the page are forced to the log disk. This may require additional log forces at the primary node if a page is very hot. This policy involves some network bandwidth overhead and some additional log disk traffic, but reduces recovery time at the secondary significantly. This policy ensures that almost no I/Os need to be done during recovery.

4 Performance Study

In this section, we analyze the various aspects of asynchronous replication to quantify the run-time overhead vs recovery time trade-offs for the three different policies described above. Since in the asynchronous replication mechanism, recovery is performed through the log, our performance results also apply to single site database recovery using log based methods such as ARIES. In Sections 4.1 through 4.5 we derive the expressions for the various recovery components and the I/O overhead. These results are then used in Section 4.6 to compare the various policies under some typical parameter settings for two classes of transaction workloads.

[5]Hot pages are those that are updated multiple times in the buffer. Cold pages are the non-hot pages.

Para.	Description	TPCA	MCW
H	# hot pages in database	10 + 1	1000
C	# cold pages in database	100000	100000
N	# total pages in database	100011	101000
n_H	# hot pages written per xact	2	2
n_C	# cold pages written per xact	1	2
n	# total pages written per xact	3	4
m	# total pages accessed per xact	3	16
Δ_d	Period of daemon wake-up for disk-writes	-	-
Δ_n	Period of daemon wake-up for network-writes	-	-
T_c	# of xacts between checkpoints	-	-
mpl	Multi-programming level		
B	Buffer size in pages on P	-	-
B_s	Buffer size in pages on S	5000	5000
$t_{logread}$	Time to read one log record	0.2 ms	0.2 ms
t_{io}	Time per data page I/O	20 ms	20 ms
$t_{logapply}$	Time to apply one log record	1.67 ms	1.67 ms

Table 3: Parameters for the Performance Model

4.1 The Model

The parameters used in the performance analysis are in Table 3. The "TPCA Value" and the "MCW Value" are the values of the parameters for two different workloads; these will be explained further in section 4.6.[6] Both the overhead as well as the recovery time are strongly determined by the buffer-write strategy. Our model assumes that dirty pages are written to disk by a daemon that wakes up periodically, so as to bound recovery time. In particular, a policy such as that mentioned in [5] is assumed whereby at a checkpoint each page that has been dirty for longer than one checkpoint interval (i.e. was updated before the last checkpoint and was not written out at the last checkpoint) is written out. In order to implement this policy, we assume that a daemon wakes up every Δ_d transactions and writes the pages that have been dirty for more than Δ_d transactions (note that we measure time in units of transactions committed). For ease of analysis, we assume that the daemon which writes the dirty pages takes a checkpoint as soon as the pages are written to disk. In order to simplify our analysis further, we also assume that the main memory available for the database buffer is large enough and that recovery bounds are tight enough that dirty pages are written out before the buffer replacement policy reclaims the buffer. It is easy to see that if n pages are dirtied per transaction, and B buffer pages are available, and if $n * \Delta_d < B$, then this will be true. This inequality is easily satisfied especially with the large main memories available today. It is easy to see that under this assumption, $T_c = \Delta_d$.

Recall our "Simple" and "Commit-And-Send" policies for secondary replica maintenance. These reflect two extremes for how frequently the secondary copy is updated. In order to model a continuum of possibilities, we introduce a parameter – Δ_n – which is the wake-up period of another

[6]The numbers for intermediate hot-cold ratios will be between those of the MCW and TPCA curves and are not shown here due to lack of space.

daemon that wakes up periodically and sends pages that have been dirty for more than one wake-up interval to the secondary across the network. In effect, our three policies can be characterized thus:

- Simple: $\Delta_n = \Delta_d$.
- Secondary Buffering: $B_s > 0$, where B_s is the amount of buffer available during normal operations at S.
- Commit and Send: $\Delta_n = 1$.

Our database is modeled as one where n_H/n accesses (and writes) go to H/N fraction of the data, modeling typical hot-set behavior. Accesses within the hot- and cold-sets are assumed to be uniformly distributed. It is easy to extend this model to incorporate multiple sets of pages with varying degrees of hotness. Also, *mpl* is the multiprogramming level, and reflects the number of active transactions at any given time. The remaining parameters of our database model are some constants for the various components of the recovery time, namely, log scan/read, data I/O, and log application (which includes both redo as well as undo phases).

With asynchronous replication, database recovery can be performed either by the primary node or by the secondary node. We will derive results for recovery both at the primary and using our three policies at the secondary. Note that results for the primary will also be valid for studying I/O overhead vs recovery time trade-offs in single site databases for log based recovery algorithms.

4.2 Number of unique pages touched in nt transactions

In our analysis, the unit for parameters such as T_c, Δ_d and Δ_n is the number of transactions executed in the given time interval. The first expression we derive is the number of *different* hot and cold pages touched as a function of the number of transactions, say nt. We have,

$$f(H, n_H, nt) = H(1 - (1 - \frac{n_H}{H})^{nt}) \qquad (1)$$

The number of cold pages accessed is given by the above equation, with H replaced by C. The number of log records written as a function of the number of transactions is, of course, given by $n * nt$.

In the next subsection we derive the various components of the recovery time, and in the subsection after that we formulate the I/O overhead.

4.3 Recovery Time

In this section, we analyze the worst-case recovery time at the primary. Worst-case analysis provides a recovery time guarantee to the user, in effect stating that if he/she is willing to suffer a $x\%$ overhead, then the system can guarantee a recovery time no more than y seconds.

The recovery time (RT) of database systems can be broke into three components: log scan time (LIO), data page reading time (DIO), and log application time ($LCPU$).

$$RT = LIO + DIO + LCPU \qquad (2)$$

LIO can be expressed as

$$LIO = (t_{logread}) * (2 * n * \Delta_d) \qquad (3)$$

the second term reflects the number of log records that need to be examined during the redo phase (Δ_d prior to the checkpoint, and Δ_d after the checkpoint).

Data I/O time is proportional to the number of data pages read during recovery, and is given by

$$DIO = t_{io} * [f(H, n_H, 2\Delta_d) + f(C, n_C, 2\Delta_d)] \qquad (4)$$

The term in [ ] reflects the total number of pages (hot and cold) touched in $2\Delta_d$ transactions, since each of these pages will be a part of DPT_R and hence will be read during recovery.

Log application time is equal to the number of log read multiplied by $t_{logapply}$, under the assumption that all log records that are read also need to be applied. This gives an upper bound on the log application time. Thus,

$$LCPU = t_{logapply} * 2 * n * \Delta_d \qquad (5)$$

With asynchronous replication, database recovery can be performed either by the Primary node or by the secondary node. The analysis so far was for recovery at the primary. Database recovery time at the secondary, on the other hand, will be different depending on the policies for keeping S up-to-date. We will now analyze the different policies.

4.3.1 Basic Policy

The recovery time at the secondary for the basic policy is likely to be very close to that at the primary, because:

- The DPTs recorded at checkpoint for the primary and the secondary are not likely to differ by more than a couple of entries ($\approx t_{roundtrip} * mpl * n/rpt$ where $t_{roundtrip}$ is the roundtrip delay for message and rpt is the average response time of a transaction), reflecting the number of pages that might be in flight and/or whose acknowledgement has not yet been received by P. For typical values of parameters, we expect this to be one or two entries and hence insignificant.
- If S subtracts from its DPT the pages that it has received since the last checkpoint, it can indeed start with a smaller DPT. However, in the analysis, we assume that dirty pages are sent to secondary only just before a checkpoint, and hence this effect is not likely to be significant.

We thus assume that the simple policy results in a recovery time at the secondary which is approximately equal to that at the primary.

4.3.2 Secondary Buffering

It is easy to see that secondary buffering can only affect DIO component of recovery (since some pages, especially hot, need not be read in from the disk) – it has no effect on LIO or $LCPU$. The (worst-case) number of pages whose latest copy the secondary has not seen is the number of different pages in $2\Delta_n$ transactions. However, an older copy of some of these pages may already exist in the buffer (especially for the hot ones) and hence need not be read from the disk. We make the following simplifying assumption: the buffer replacement policy at S can discriminate between hot and cold pages, and hence the buffer will first contain the hot pages, and only if further space is available, will it contain the cold pages. In that case, the number of I/O's required during recovery at S is given by:

$$I/O_S = \begin{cases} f(C, n_C, \Delta_d) * (1 - \frac{B_S - H}{C}) & if B_S > H \\ f(C, n_C, \Delta_d) + \\ \quad f(H, n_H, \Delta_d) * (1 - \frac{B_S}{H}) & otherwise \end{cases} \quad (6)$$

Thus,

$$DIO = t_{io} * I/O_S \quad (7)$$

This follows from the assumption that if the buffer size is enough to accommodate all the hot pages, then I/O's need to be done only for the cold pages, and this is given by the number of cold pages required multiplied by the probability of not finding a cold page in the buffer. On the other hand, if the buffer is not sufficient to store the hot pages, then I/O's will be required for a fraction of the hot pages, and for all the cold pages.

4.3.3 Commit and Send

The commit and send policy reduces DIO time as well as the log scan time and the log application time. The data I/O time can be reduced close to zero when $B_s > f(C, n_C, \Delta_d) + f(H, n_H, \Delta_d)$. Assuming that the secondary node has enough memory to keep all dirty pages at any given point in time, all such pages except those that are in-flight when the primary fails will be in memory. Therefore, in equation (2) DIO will be $mpl * n_C$. As a worst case assumption and a simplification, we will assume that the number of log records that need to be examined will be the same as in the case of the primary analysis and hence LIO and $LCPU$ are the same as in equations 3 and 5.

4.4 Write I/O Overhead Analysis

In this section, we will derive results for the I/O overhead due to page writes both on the primary and the secondary. The more frequently the dirty page daemon wakes up and schedules its writes, the higher the I/O overhead but smaller the recovery time.

First we will analyze overhead at the primary. It is easy to see that when a page is dirtied, it is either written at the immediately succeeding daemon wake-up, or at the one after that. Let us define the following terms, in order to determine the amount of write I/O that is generated by this policy:

P_a	Probability that a given page is updated in a time unit of Δ_d
P_w	Probability that a given page is written to disk due to a daemon wake-up

A given page is written out by the dirty page daemon at its $i+1$th wake-up if and only if:

- it was updated in the interval between the $i-1$th and the ith wake-ups
- it was not written out at the ith wake-up

The two events above are independent events. Also, the probability that a given page was not written out at the ith wake-up is $(1 - P_w)$. Based on this, we have the following equation:

$$P_w = P_a(1 - P_w) \quad (8)$$

because the first term in the right hand side reflects the probability that the same page was accessed in the interval $i-1$ to i, and the second term reflects the probability that it was not written out at wakeup i. Solving this for P_w, we get

$$P_w = \frac{P_a}{1 + P_a} \quad (9)$$

If there is a hot- and a cold-set, then P_w can be broken up into P_w^H (expressed in terms of P_a^H – the probability that a hot page is accessed in an interval of length Δ_d), and P_w^C, (expressed in terms of P_a^C). Now, $P_a^H = f(H, n_H, \Delta_d)/H$, and a similar expression exists for cold pages. We can thus write an expression for G, the total write traffic to disk between two checkpoints as:

$$G(\Delta_d) = [H * P_w^H + C * P_w^C] \quad (10)$$

and for W, the total write traffic per transaction as:

$$W(\Delta_d) = G(\Delta_d)/\Delta_d \quad (11)$$

assuming that H, C, n_H and n_C are constants.

Now let us analyze I/O overhead at the secondary. In the simple policy since $\Delta_n = \Delta_d$ and there is no buffering at the secondary, I/O overhead at the secondary is the same as at the primary. Under the assumption that the secondary has enough buffers (secondary buffering and "commit and send" policies) to store pages dirtied in $2\Delta_d$ transactions, the normal write traffic to secondary's disk is also given by $G(\Delta_d)$, independent of Δ_n. (Remember that in all three policies $\Delta_n <= \Delta_d$.) This follows from the fact that the secondary, if it has this buffer space, can mimic the primary's disk write policy exactly, forcing a page to the disk only when the primary does the same. Thus, all three secondary policies have the same I/O overhead as on the primary and is given by equation 11.

4.5 Network Overhead

The daemon that sends dirty pages to the secondary follows exactly the same policy as the daemon which does the disk writes. Thus, it is easy to see that the network (bandwidth) overhead is simply given by $W(\Delta_n)$, where W is expressed in equation 11.

4.6 Performance Results

Based on the analysis in the previous sub-sections, we will plot the behavior of the performance measures of interest in the next two sections. We will use one workload based on the **TPCA** benchmark [7]. However, the TPCA benchmark represents small transactions and is only one of the various kinds of workloads found in practice. Also it has a very small hot set; thus it is hard to study policies such as secondary buffering. We will use another workload based on a study by [17] to represent a medium complexity workload (**MCW**) with a bigger hot set. The parameters used in TPCA workload and MCW are shown in Table 3. The TPCA workload has actually been analyzed as three sets of pages (ACCOUNT, TELLER and BRANCH) with different degrees of hotness. Note that the total number of database pages, and the hot and cold pages per database are given per node.

Our goal is to understand recovery time vs I/O overhead trade-offs. However, since the plot of recovery time vs I/O overhead is hard to understand directly, we will first plot each of these against the independent variables Δ_d and Δ_n, and then eliminate the independent variables. In the following, we assume that $\Delta_d = \Delta_n$ for the Secondary Buffering policy.

4.6.1 Write I/O Overhead

Without replication, there would be no write I/Os at the secondary and thus these constitute the I/O overhead due to replication. Figure 1 shows the write I/Os at the secondary as a function of the Δ_d interval (as given by equation 11). As explained above, this is the same as the number of write I/Os at the primary. The line for TPCA has a very sharp knee because the TPCA hot set size is very small; in our analysis, we have 100,000 ACCOUNT pages, 10 TELLER pages and 1 BRANCH page per node. Thus the size of the hot page set is 11 against a cold page set of 100,000. Once $f(H, nt, \Delta_d)$ approaches H, increasing Δ_d further does not reduce the write I/O overhead significantly. On the other hand MCW has a hot set size of 1000 pages and thus the number of write I/Os tends to fall off much more smoothly.

4.6.2 Recovery Time

Expressions for recovery time on primary and on the secondary with all three policies have been derived in section 4.3. Figure 2 shows the behavior of the TPCA workload against Δ_d for the three different policies. Note that the lines for the "simple" policy and for "secondary buffering" policy overlap except for small values of Δ_d. Recall that the secondary buffering policy enabled the hot set to reside in the memory of the secondary and hence there was no need for I/O during recovery if the primary failed. If the hot set is small, this does not offer a significant advantage since the number of I/Os that need to be done for the cold set greatly exceed that for the hot set. Thus, secondary buffering does not offer much advantage for a workload such as TPCA which has a very small hot set. Figure 3 shows the same parameters for MCW. Since the hot set is large in this case, secondary buffering shows a significant improvement over the basic policy. Note that there is a substantial recovery time improvement for the commit and send policy for both workloads.

4.6.3 I/O Overhead vs Recovery time

Figure 4 shows the recovery time as a function of I/O overhead for the three secondary recovery policies for the TPCA workload. This is simply the result of eliminating Δ_d between Figures 1 and 2. Figure 5 shows the same for MCW. The commit and send policy improves performance markedly for MCW, but the improvement is not as significant for TPCA. For example, if one is willing to suffer 1.1 write I/Os per TPCA transaction there is hardly any difference in performance for the three policies (Figure 4). Note that for TPCA since the ACCOUNT relation is in the cold set, each TPCA transaction must do a minimum of one I/O. Thus it is possible to reduce the two potential write I/Os for the hot BRANCH and TELLER updates to 0.1 I/Os per transaction, even in the case of the basic policy without increasing recovery time by more than 5 seconds.

For MCW, such an optimization is not possible for the basic policy and hence there is scope for the "secondary buffering" and the "commit and send" policies to improve performance. Thus, the conclusion is that "simple" is good enough for TPCA-like workloads, with small hot set sizes. However, for MCW-like workloads, "commit and send" seems to be best.

Figures 4 and 5 can also be re-interpreted to study tradeoffs between recovering at the primary and recovering at the secondary. If the primary fails because of a software failure (kernel or DBMS bug), the node manager must choose to either ask secondary to assume primary role after recovery or wait for the primary to be rebooted or the DBMS to be restarted. For TPCA, the only difference between recovering at the failed primary ("simple" curve) and at secondary is the primary reboot or DBMS restart time (which can be anything between 20 seconds to 2 minutes). However, for MCW, "commit and send" can reduce recovery time by a significant fraction.

4.6.4 Network Overhead

The shape of the network overhead against Δ_n curve is shown in Figure 6 for TPCA (the figure for MCW is not presented for lack of space; see [2] for details)[7]. Since the TPCA curve falls very sharply with Simple and Buffering policies, it is possible to minimize network overhead to a large extent by using reasonably small values of Δ_n. Note that the "commit and send" policy requires a lot more network bandwidth than the other two; however, when the network bandwidth is not the bottleneck, this might be a reasonable price to pay for the improved recovery time. With emerging optical technologies, this is more likely to be the case in the future.

5 Comparison with Other Schemes

In this section, we demonstrate that the run time I/O and CPU penalties in our algorithm are lower than other high availability schemes for TPCA transactions, but the network overhead is higher.

The overheads for the different schemes for a single TPCA transaction are shown in Table 4, assuming "simple" policy for asynchronous algorithm. The I/O is assumed to be only due to the ACCOUNT pages, since BRANCH and TELLER pages are written very infrequently.

In our algorithm, the secondary can schedule a disk write operation without reading the original page image from disk because the entire page image is shipped across the network. Thus, in our algorithm, an I/O overhead of 1 extra write is incurred. In each of the other three schemes (log spooling [11, 9], RADD [14] and the synchronous schemes which run transactions on both the replicas [16]), the secondary needs to perform a read before a write. This results in an overhead of two I/O's per transaction, which is 100% larger than our scheme.

The numbers in the CPU overhead column are based on the following instruction counts: a) 8 K (roundtrip) per message, b) 5 K for starting an I/O, c) 250 K per TPCA transaction, and d) 8K for applying each log record. The synchronous scheme pays the maximum penalty, because the transaction has to be executed on both nodes, and also involves commit processing which is assumed to be 2 messages per transaction.

The numbers in the network bandwidth column are based on a 4K byte page, 200 bytes per commit message, and 200 bytes per log record.

[7]In reality, the Commit and Send curve is just a point, since it assumes that $\Delta_n = 1$, however for clarity of exposition, we draw a horizontal line.

	I/O	CPU (K inst.)	Network Bandwidth
Asynchronous (simple)	1	8 (log send) + 8 (ACCOUNT to S) + 5 (ACCOUNT I/O at S)	4 KB
Synchronous (logical)	2	250 (xact at S) + 2*8 (commit processing)	0.6 KB
Log Spooling	2	3*8 (log apply) + 10 (ACCOUNT I/O at S)	0.6 KB
RADD	2	8 (ACCOUNT to S) + 10 (Parity I/O at S)	4 KB

Table 4: Overheads in Replication Schemes

6 Conclusion

In this paper, we presented the details of an asynchronous replica management mechanism for handling software and hardware failures in a Shared Nothing database machine environment. Our goal was to provide fault tolerance without sacrificing performance in the failure free mode of operation. Our mechanism updates the secondary replica asynchronously by sending dirty pages before they are discarded from primary's buffer. Log records for all nodes are stored at a log server, which is hardened against failures. This log is symmetrical with respect to the primary and secondary, because it records the disk state of both replicas. If a primary node fails, the secondary uses the log to bring itself to the latest transaction-consistent state.

We presented three policies which sent dirty pages to the secondary with different frequencies. A performance model was presented that helped us analyze the resulting recovery time vs I/O and network overhead trade-offs for these policies. We showed that aggressive policies such as "commit and send" pay off for workloads such as MCW, but not for TPCA. We also showed that recovery at secondary can in fact be faster than that at the primary. This is mainly because the secondary buffers are not lost when the primary node goes down, resulting in reduced data I/O.

Finally, we showed that our algorithm has lower I/O and CPU overheads, compared to other schemes for high availability like log spooling, RADD and synchronous replication.

Acknowledgements

The authors would like to thank C. Mohan for pointing out some flaws in our initial design; George Copeland for general discussions; and the reviewers for their many insightful comments.

References

[1] Bhide, A., Goyal, A., Hsiao, H., and Jhingran, A., "Asynchronous Replica Management for Shared Nothing Architectures" IBM TJ Watson Tech Report RC 16403, Dec. 1990.

[2] Bhide, A., Goyal, A., Hsiao, H., and Jhingran, A., "An Efficient Scheme for Providing High Availability" IBM TJ Watson Tech Report RC 17571, Jan. 1992.

[3] Copeland, G., Alexander, W., Boughter, E., and T. Keller, "Data Placement in Bubba," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[4] DeWitt, D., Ghandeharizadeh, S.,Schneider, D., Bricker, A., Hsiao, H, and Rasmussen, R, "The Gamma Database Machine Project," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[5] Gray, J., "DISC," A talk given by Jim Gray at University of Wisconsin, Madison, February 1989.

[6] Gray, J., Horst, B., and Walker, M., "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput," Proceedings of 16th VLDB Conference, Australia 1990.

[7] Gray, J., Editor, "Benchmark Handbook," Morgan Kaufmann Publishing 1991.

[8] Jhingran, A. and Khedkar, P., "Analysis of Recovery in a Database System Using a Write-Ahead Log Protocol," Proc. ACM SIGMOD, June 1992.

[9] King, R., Garcia-Molina, H., Halim, N., and Polyzois, C., "Management of A Remote Backup Copy for Disaster Recovery," University of Princeton CS-TR-198-88

[10] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P., "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", To appear in ACM TODS, March 1992.

[11] Mohan, C., Treiber, K., and Obermarck, R., "Algorithms for the Management of Remote Backup Data Bases for Disaster Recovery," IBM Research Report, July 1990.

[12] Patterson, D., Gibson, G., and Katz, R., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," Proceedings of the ACM-SIGMOD International Conference on Management of Data, Chicago, May 1988.

[13] Stonebraker, M., "The Case for Shared Nothing," Database Engineering, Vol. 9, No. 1, 1986.

[14] Stonebraker, M. and Schloss, G., "Distributed RAID - A New Multiple Copy Algorithm," Proceedings of the 6th International Conference on Data Engineering, Los Angeles, February 1990.

[15] Tandem Database Group, "NonStop SQL, A Distributed, High-Performance, High-Reliability Implementation of SQL," Workshop on High Performance Transaction Systems, Asilomar, CA, September 1987.

[16] Teradata, "DBC/1012 Database Computer System Manual Release 2.0," Document No. C10-0001-02, Teradata Corp., NOV 1985.

[17] Yu, P.S et al., "Coupling Multi-Systems Through Data Sharing," Proc. of the IEEE 75(5), May 1987.

Figure 1:I/O Overhead for TPCA and MCW

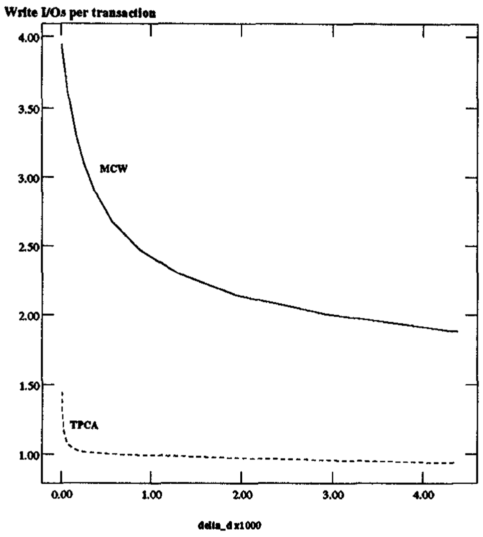


Figure 2:Recovery Time for TPCA

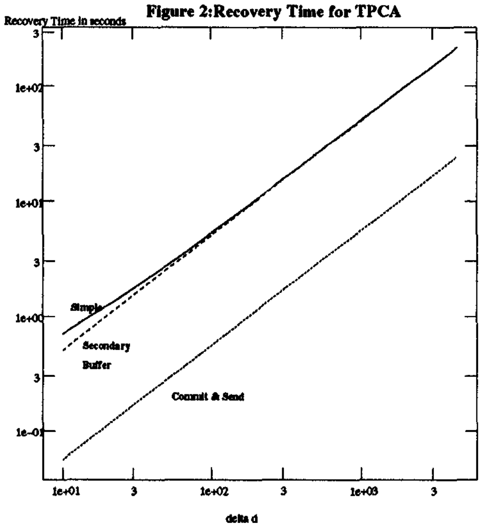


Figure 3:Recovery Time for MCW

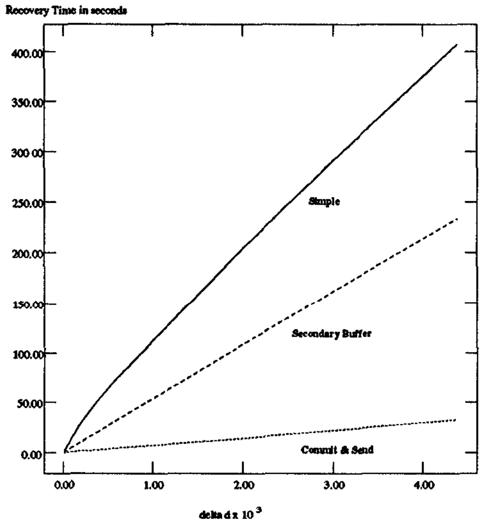


Figure 4:Recovery Time vs. I/O for TPCA

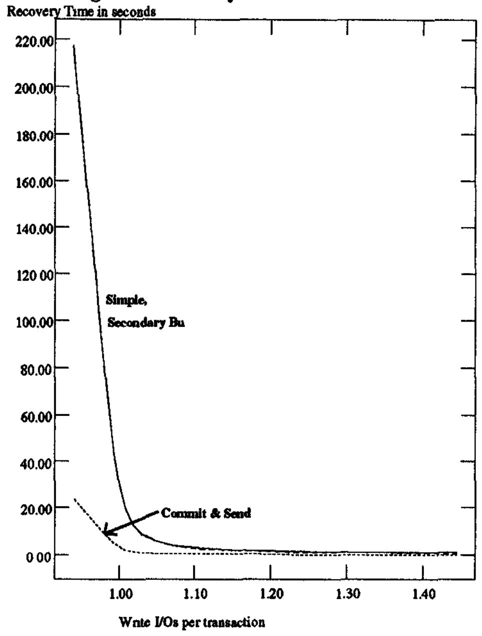


Figure 5:Recovery Time vs. I/O for MCW

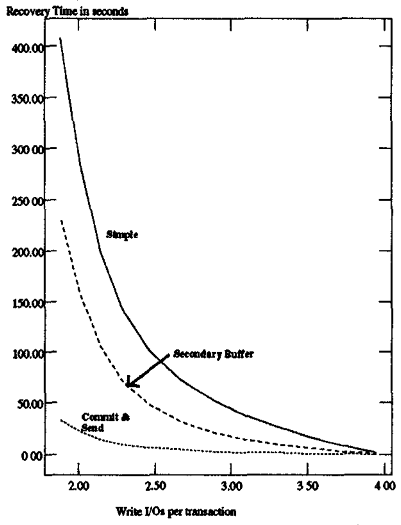
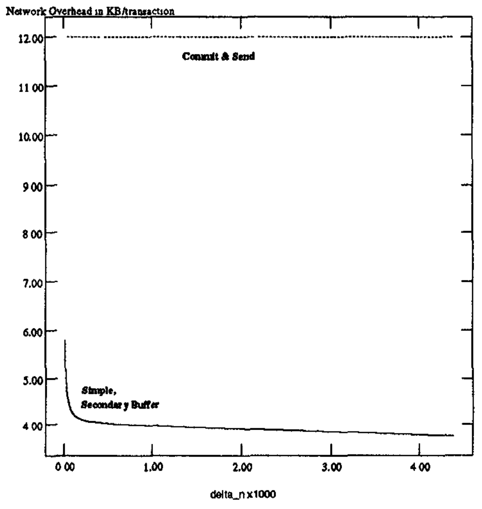


Figure 6:Network Overhead for TPCA


Network Overhead in KB/transaction
Commit & Send
Simple,
Secondary Buffer
delta_n x1000