RECOVERY IN A TRIPLE MODULAR REDUNDANT DATABASE SYSTEM

Frank Pittelli
Hector Garcia-Molina

CS-TR-076-87

January 1987

Recovery in a Triple Modular Redundant Database System

Frank Pittelli

Computer Science Department
United States Naval Academy
Annapolis, MD 21402-5002

Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, N.J. 08544

ABSTRACT

In a Triple Modular Redundant (TMR) database system the database is fully replicated at three computers. All transactions are executed at all nodes in the same relative order. The system can tolerate the arbitrary failure of a single computer since the correct data can be obtained from the two operating copies. After a failure, it is important to repair the computer so that the system can tolerate additional future failures. Repair in this case involves getting a correct and up to date copy of the database, without halting the two operational nodes. In this paper we analyze this database recovery problem. We describe a solution that has been implemented on an experimental TMR system running on SUN-2/120 workstations. We also present performance results that illustrate the cost of recovery.

This work has been supported by NSF Grants DMC-8351616 and DMC-8505194, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Concurrent Computer corporations.

December 5, 1986

Recovery in a Triple Modular Redundant Database System

Frank Pittelli

Computer Science Department
United States Naval Academy
Annapolis, MD 21402-5002

Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, N.J. 08544

1. INTRODUCTION

Triple Modular Redundancy (TMR) is a strategy for protecting against arbitrary hardware failures. Given a task, a TMR system will execute it three times, in parallel, on independent hardware elements. The results of the three executions are compared by a voter. If two or three of the results agree, they are taken to be the correct output for the task. TMR provides a high degree of reliability because it can tolerate an arbitrary failure of one of the processing elements.

Typically, TMR has been used at the component level, where each processing element is a chip or a bus [Siew82]. The voting is done by a hardware voter that is assumed not to fail. Furthermore, the inputs for the task (e.g., the signals that go to each chip) are also distributed by error-free hardware.

More recently, it has been suggested that TMR can be used at the application level directly, with all coordination between processing elements done entirely by software [Lamp84a, Schn82, Garc86]. In a database application, for example, each processing element would be a complete computer with a copy of

the entire database. Input transactions would be distributed to all computers by a software agreement protocol so that they all execute the same sequence of transactions. The outputs of each execution would be sent to the users, who would then take the majority result as the correct one.

Such an approach has several advantages. Off-the-shelf computers can be utilized, as opposed to the special purpose circuitry required by component level TMR. Communication costs can be reduced since only short, high level transactions (and their results) are exchanged, as opposed to every single signal that a chip uses. Given the reduced communications, it is easier to physically isolate the processing elements. This makes it less likely to have a failure of two or more elements due to physical problems (i.e., power failure, fire, etc.). Of course, both component and application level TMR require triple hardware, but that is the price to pay for such a high degree of reliability.

When used for database processing, application level TMR has complications that component TMR does not. In database processing there is a substantial state (mainly the database) that has to be maintained from one task to the next. When a computer fails, all or part of its database can be destroyed. At recovery time, the lost data, including any data modified by the operational nodes during the failure, must be copied from the other nodes. Clearly, if the operational nodes simply halt processing during this "exchange period" no useful work will be accomplished by the TMR system. Consequently, the other nodes must continue processing new transactions. So how can the recovering node obtain a valid copy of this "moving target"? Furthermore, even if it can acquire a consistent copy, how much degradation will be caused by the recovering node? That is, how much extra load is placed on the other nodes when all or part of the database is transferred over the network? There is also an interaction between the fraction of the database that can be lost and the reliability provided by the system. As the fraction grows, the recovery time also grows. The longer this period, the more vulnerable the system will be to a second failure that can cripple the entire system. What exactly is the relationship between these and other system parameters?

In this paper we address these questions. We start, in Section 2, with a

brief overview of our TMR database system and its operation during no-failure periods. (Due to space limitations, a number of simplifications will be made. A detailed description of normal operation can be found in [Pitt86], while the theory of TMR database processing is presented in [Garc86].) Then, in Section 3, we describe a strategy for recovering from arbitrary failures in a TMR database system and study its performance during recovery periods (Section 4). Both the design and results presented here are based on an experimental database TMR system implemented at Princeton on a collection of SUN-2/120 workstations.

Note that the TMR recovery algorithms we will present can be generalized easily to N-Modular Redundancy. However, we will restrict our discussion to TMR both because it is the most likely in practice and because our experimental results are for such a level of replication.

2. NORMAL OPERATION

One important advantage of a TMR system is that it makes a simple design feasible. All nodes perform the same functions and maintain the same data. Furthermore, normal processing at each node breaks down naturally into three quite independent processes: transaction scheduling, transaction processing, and failure detection.

To describe the operation of these components we trace the execution of a single transaction through the system. Figure 1 illustrates this process, with each column representing the processes within a single node. A user submits a transaction from his own process, possibly running on a computer not involved in the TMR system. The transaction arrives at one of the three scheduling processes (step 1 in Figure 1). The scheduler forwards the transaction to its cohorts on the other nodes (step 2), following a Byzantine Agreement synchronization protocol [Peas80, Dole82, Lamp82, Lync82, Fisc83, Cris84]. This protocol generates, at every non-failed node, the same sequence of transactions, $T_1, T_2, \cdots$, with corresponding timestamps, $t_1, t_2, \cdots$ (The synchronization protocol uses physical clocks that are synchronized within some maximum tolerance. Algorithms that perform such clock synchronization are described in

[Lund84, Lamp84b, Dole84].)

It can be shown that the transaction scheduling protocol operates correctly in the face of a single, arbitrary node failure (a communication failure can be viewed as a node failure). The non-failed nodes will produce exactly the same transaction schedule. Furthermore, T_i will be added to the schedules before time t_i.

As each scheduler generates the sequence of transactions it is sent to the transaction scheduler which executes each transaction in the order given (step 3). When the transaction manager executes a transaction, it uses a conventional, single-node database management system. There is no need to request remote locks or do any further distributed synchronization. The only non-conventional aspect is that transactions must be committed in the pre-agreed upon order.

Once the transaction completes, the results are sent to the user and a database "signature" is sent to the failure detection processes (or voters) in step 4. (We will discuss signatures further in the next section.) Each voter is in charge of comparing the databases in an attempt to determine if its own transaction manager has produced a result that differs from the majority. If this happens, some type of failure has occurred and the local recovery process is initiated. (Note, the voter does not care what caused the failure, only that a failure has occurred.) Keep in mind, in a TMR environment each node must be responsible for detecting its own failure, because the other nodes can never be fully trusted.

3. RECOVERY

The design of any recovery algorithm begins with an understanding of the types of failures which are possible. However, in the environment that we have chosen, any type of behavior is possible, therefore, we can not list all the failures. Rather, we must define failures in terms of their effect on the local database copy. That is, if a given database page is different from that maintained by the majority of nodes, then one or more failures have occurred locally. No matter what the cause, the page is considered corrupted and must be

replaced.

Clearly, during an arbitrary "failure period", a given node may never detect that is has corrupted its own database. In this case, the node will continue to execute transactions based on incorrect data which may corrupt other parts of the database. Unfortunately, there are no mechanisms to prevent such destructive behavior during a failure. We must hope that, at some point, the node repents and begins to function properly. That is, the node begins to execute its algorithms correctly and detects the inconsistency of its own database. When that point is reached, the failure period ends and the "recovery period" begins.

3.1. A Recovery Model

The actions required to recover from a node failure can be generalized by a recovery model. At this point, we simply present our model followed by a discussion concerning the implementation that we have chosen. (A more detailed examination of various implementation issues can be found in [Pitt86].)

Figure 2 shows the recovery model as a sequence of actions. The cycle begins when one of the nodes corrupts its local database copy (Failure). Then, by examining the database copies of the other nodes, the failed node can detect its *own* failure and can begin its recovery strategy (Detection). Remember, any number of transactions can be executed in between failure and detection, and the inconsistency may be corrected before it is even detected. Once a failure is detected, the node must construct any consistent copy, called the *snapshot*, and execute all transactions scheduled after that point. We emphasize the phrase "any consistent copy", because the node may choose it from the past, present, or future. (One of the many alternatives will be discussed shortly.)

In general, snapshot acquisition is a two-phase operation. First, the recovering node must request some information from the other nodes to guarantee consistency (Snapshot Request). Once the information is received, that node can install the consistent database copy (Snapshot Installation). At that point, the recovering node can begin processing all the transactions which have timestamps greater than the snapshot.

At this time we should stress a fundamental difference between the Detection and Snapshot Request phases. The Detection phase must be executed periodically to guarantee consistency. Furthermore, the amount of time between these periodic checks directly affects the probability that the entire TMR system will fail. That is, a single node failure must be detected and corrected before another node fails, otherwise all database copies may become inconsistent. Therefore, detection must be accurate enough to catch most failures, frequent enough to safeguard the system, and simple enough so as not to degrade normal transaction processing. On the other hand, the Snapshot Request phase is executed only when a failure has occurred and it must acquire enough information to reconstruct a consistent snapshot. That is, snapshot processing is driven by the frequency of failures and must do substantially more work than detection.

Now, our attention turns to some practical details associated with different implementations of this recovery model.

3.2. Database Signatures

Implementation of the Detection and Snapshot Request phases is facilitated by the use of *database signatures.* We define a signature to be any data item which "summarizes" the database. For example, a conventional redo log can be used as a database signature because it reflects all of the changes made to some version of the database. Such signatures can be used to detect failures (during the Detection phase) and/or to determine the set of corrupted pages (during the Snapshot Request phase). As usual, database signatures can be constructed in many ways, each with their own advantages and disadvantages [Lipt84, Fuch86, Park83, Metz83]. These methods vary mainly in the amount of storage, CPU, and network resources used, but they all have the same goal: to compare two databases as efficiently as possible.

In our implementation, failure detection is accomplished using a four-byte checksum for the entire database. This checksum is easily maintained in memory and is the only information exchanged with the other nodes, thereby keeping network traffic low. The detection checksum itself is constructed from

a set of four-byte checksums for each page. During the Snapshot Request phase, the nodes exchange this larger set of checksums to pin-point those pages that have been corrupted and must, therefore, be exchanged during the Snapshot Installation phase. Naturally, the set of checksums can be exchanged iteratively, using a tree of checksums, to determine the corrupted pages.

3.3. The Care and Feeding of Snapshots

If the Detection and Snapshot Request phases are performed separately (as in our implementation), the selection of an appropriate snapshot time becomes important. If it is chosen in the past, for any node, that node must maintain multiple versions of its local database copy. Fortunately, there are techniques which can be used to satisfy this goal, trading off storage and CPU costs [Adib80, Lind86]. However, this approach places most of the burden on the non-recovering nodes. That is, the nodes which didn't fail must either maintain old versions or reconstruct them when requested. On the other hand, if the snapshot time is chosen in the future, for all nodes, then they can simply wait until the designated time and "freeze" a single copy of the database. As before, conventional techniques exist to perform this act efficiently, while allowing the nodes to continue transaction processing [Lind86]. In either case, all nodes must "agree" on the snapshot time during the Snapshot Request phase. This time must be selected far enough into the future so that each node can store the snapshot before executing any transactions with later timestamps. Furthermore, it must be early enough so as to assure quick recovery from the failure. (Recall, all nodes execute the same sequence of transactions, but they proceed at their own rate. Consequently, each node never really knows what transactions have already been executed by the other nodes.)

In our implementation, we rely on the schedulers to determine the earliest possible snapshot time, and use a simple versioning scheme for the snapshots themselves. Figure 3 shows the steps taken by all nodes during the Snapshot Request and Installation phases. To start the process, the voter informs the local recovery manager of a failure (Step 1). In turn, the recovery manager submits a snapshot transaction to the local scheduler (Step 2). The snapshot transaction, like any other, is assigned a position in the global schedule and is

eventually processed by the transaction managers on all nodes (Steps 3 and 4). (Note, if the recovering node doesn't schedule the snapshot transaction, or can't execute it, it isn't ready to recover anyway.) When a snapshot transaction is executed, the transaction manager stores a snapshot of the local database. Of course, these snapshots are taken at different physical times, but they represent the database seen by each node after the same transaction. Once the snapshot has been taken, each transaction manager informs the local recovery manager and resumes normal transaction processing (Step 5). (Actually, the transaction manager on the recovering node may have to postpone processing. More on this shortly.) At this point, the failed recovery manager sends it's signature to both of the non-recovering nodes, who independently determine which snapshot pages should be sent to the recovering node (Step 6). The recovering node compares corresponding pages as they are received and installs them in the local database if they agree. When the other nodes have finished sending pages, the recovery manager allows the local transaction manager to resume transaction processing (Step 7).

Why do both non-recovering nodes send the necessary database pages during a recovery? After all, if we are assuming that only one failure can occur at a time, can't the recovering node "trust" a single node to send the correct information? The recovery algorithm would, in fact, work properly if only one non-recovering node participated. However, by comparing the pages from two nodes, we have added an extra level of protection. That is, if a second failure had already occurred, the recovering node would detect this, and could halt before additional problems were generated. However, since it is outside this paper's scope we will not address this issue further. Suffice it to say that, even if we assume certain properties for the system, it may be beneficial to periodically guarantee that those properties hold.

Finally, transaction timestamps are used to simplify snapshot generation. In particular, every page in the database is marked with its most recent update time. At any time, each page has exactly one "snapshot copy", which is never modified, and possibly a "work copy". The snapshot copy has a timestamp less than or equal to the current snapshot time, while the work copy has a timestamp greater than the snapshot time. If no work copy exists when a

transaction needs to access it, one is created by copying the snapshot copy. In this way, multiple copies only exist for those pages accessed since the last snapshot. Furthermore, when a new snapshot is requested it can be generated simply by advancing the snapshot time. When eventually accessed, the old work copies will become the new snapshot copies, and the old snapshot copies will be deleted.

For example, suppose that one page copy exists with a timestamp of 0900 and that the current snapshot time is 0930. If a transaction with timestamp 0945 attempts to access that page, a new work copy, with timestamp 0945, will be created. Any number of transactions can access this work copy, each of which advances the work copy's timestamp. Assuming that another snapshot is requested at time 1100, it is accomplished simply by advancing the current snapshot time. Now, if a transaction accesses the page, it will find two copies, both with timestamps less than the current snapshot time. In this case, the page with the oldest timestamp is deemed the snapshot copy and is copied into the older page, which becomes the new work copy.

3.4. Transaction Processing During a Recovery

Given that the Snapshot Request and Installation phases take some amount of time, we must discuss how transaction processing is affected during that period. As stated before, once the non-recovering transaction managers have created their snapshots and informed their local recovery managers, they are free to continue normal processing. The recovery managers are responsible for exchanging the necessary signatures and pages. With that in mind, the only impact on the non-recovering nodes is the work required to execute the snapshot transaction itself and the overall degradation caused by the recovery manager's work.

On the other hand, the transaction manager on the recovering node should postpone transaction processing until the end of the Snapshot Installation phase to prevent transactions from accessing corrupted pages. Consequently, it will "lag behind" the other nodes for a period of time, known as the *catch-up* time. (Actually, the recovering node can continue to process transactions but the

recovery algorithm becomes too complex to discuss here.) That is, the recovering node will be sending results to the user and detection signatures to the voter some time after the other two nodes. As long as no other node fails, this catch-up period doesn't really affect the voters or the user. Rather, they will see a majority of results from the first two nodes and will simply ignore the third when it arrives. On the other hand, when another node fails, the user and the voters will suffer a delay while the previously recovered node catches up. Keep in mind, the TMR system can cope with a failure "any time" after Snapshot Installation, it's just that the recovering node has to finish its backlog of transactions before it can take the next snapshot.

4. PERFORMANCE DURING RECOVERY

One of the major advantages of a TMR design is that the system can continue transaction processing while a single node is recovering from a failure. However, since the non-failed nodes must help in the recovery process, it is important to understand the load imposed by snapshot processing. To study this issue we designed a set of experiments on an experimental TMR database system. The system consisted of three Sun-2/120 workstations running Sun's version 2.0 of UNIX (based on Berkeley 4.2BSD UNIX), connected by a 10 Mbit/sec ethernet. Each workstation contained two disks, one supporting the operating system and one supporting the database. Interprocess communication was performed using the UDP/IP facilities of Sun UNIX. Finally, all user transactions originated from a fourth SUN workstation. This workstation generated new transactions at a steady input rate I ranging from 5 to 17 transactions per second. Each TMR processing node received a third of these transactions for scheduling.

For experimental purposes a number of simplifications were made to the database system. In particular, transactions were synthetically generated using probability distribution functions, not by "real" applications. Also, the computers were interconnected using an ethernet, not direct connections (its the only network we had available). Finally, to study performance during recovery, one of the transaction managers was modified to periodically report an artificial failure. The recovery managers, in turn, exchanged a fixed number of pages,

from 0 to 50, known as the *exchange size*, during snapshot processing. Keep in mind, despite these simplifications all the TMR scheduling, failure detection, and failure recovery algorithms were carefully implemented, so we believe the performance results are realistic. For example, even though we had an ethernet, which is a single point of failure, our design did not utilize the broadcast capability of the ethernet. Hence, we believe that a different network would not change our performance results significantly.

Throughout our experiments, four measurements were of interest. First, the average transaction response time during a recovery was measured. In particular, the response time for every transaction executed after a failure was reported, and before the recovering node had fully caught-up, was averaged. The second parameter was the time between the initial snapshot request and the completion of the snapshot installation, known as the *recovery response time*. Lastly, the number of transactions postponed by the transaction manager (the *queue size*) and the amount of time to empty that queue (the *catch-up time*) were monitored. This cycle of failure, recovery and catch-up was continued until a sufficient confidence level was achieved.

4.1. Results

Examination of the average transaction response time provides a good idea of how snapshot processing degrades the overall system. Figure 4 shows this measurement for normal processing and for recoveries involving exchange sizes of 0, 2, 10, 20, and 50. It is evident that snapshot transactions affect the system, even when no pages are exchanged ($e0$). The response of the normal system is never more than 200 ms, while the snapshot curves are rarely lower than 300 ms. We conclude that the number of messages exchanged during the recovery algorithm (recall Figure 3) hinders the system's ability to process transactions efficiently.

It is interesting to note that the response times for $e0$ and $e2$ decrease at first, as the input rate increases. This effect is caused by the "efficiency" of recovering from small failures. In particular, at low input rates the system is under-utilized and snapshots are performed quickly. Therefore, only a few

transactions are processed during the recovery and they are delayed heavily by the snapshot transaction itself. As the input rate increases, more transactions are processed by the system during the recovery. The first few of these experience large delays due to the snapshot transaction, while the others experience short delays caused by snapshot installation. Consequently, the "average" delay for a transaction is smaller, until a point at which the response times begin to increase again because of the effects of a high input rate.

We should also note that the response curves for exchange sizes of 20 and 50 ($e20$ and $e50$) are roughly the same. In other words, when the time for snapshot installation greatly exceeds that for the snapshot transaction, the average transaction response time is degraded "only so much". Transactions processed during the exchange of database pages are delayed by a constant amount, caused by the additional network traffic. If more pages are exchanged, more transactions are processed, but each experiences the same increase in response time.

The increasing snapshot installation times (i.e., recovery response times) mentioned in the previous discussion are shown in Figure 5, which plots it for a number of exchange sizes. The curve for an exchange size of 0 ($e0$) shows the time required to schedule and execute a snapshot transaction itself. Comparing it to curve $e50$, we see that snapshot installation can easily require more time than snapshot transaction processing. Furthermore, if we plot the recovery response time versus the exchange size, for a given input rate, we would see a straight line. That is, snapshot installation is a linear function of the number of pages exchanged and, as shown in Figure 5, relatively independent of the input rate.

Turning now to the queue size, shown in Figure 6, we see that it increases as the input rate increases, but is affected most by an increase in the exchange size. Quite simply, the greater the number of pages exchanged, the greater the number of transactions that must be postponed. This concept is neither new, nor unusual, and will not be discussed further.

Finally, we must answer an important question; does it take longer to install a consistent database snapshot or to catch-up to the other nodes after

such an installation? The answer is shown in Figure 7, which plots the average catch-up time for various exchange sizes and input rates. We see that for low input rates, a recovering node can always catch-up faster than it can exchange the database pages (compare Figures 5 and 7.) Even for high input rates the catch-up time is comparable to the recovery response time.

In fact, the catch-up time (C), in seconds, can be predicted using a simple formula based on the transaction manager's maximum throughput (M), in transactions per second, the input rate (I), in transactions per second, and the queue size (Q), in transactions. In particular, we have

$$C = \frac{Q}{M-I}$$

That is, the catch-up time equals the number of transactions queued divided by the excess processing power available ($M-I$). For example, given an exchange size of 20 pages and an input rate $I = 10$ transactions per second, we see from Figure 6 that $Q = 15$ transactions. Furthermore, other TMR experiments (not discussed in this paper) showed that the transaction manager can process at most $M = 40$ transactions per second. According to our formula, we have

$$C = \frac{Q}{M-I} = \frac{15}{40-10} = 0.5 \text{ seconds}$$

As a check, this value corresponds to the catch-up time shown in Figure 7 for the given exchange size and input rate. (A similar formula can be derived which predicts the queue size when the exchange size is known.)

4.2 Discussion

The results of Figures 4 through 7 illustrate the relationship among the various system parameters. The next question to consider is how to use this information to design a system that gives users the reliability and performance they expect.

Before addressing this question, we make one clarification. Our performance experiments studied fully disjoint recoveries. That is, we assumed that a node had fully recovered from a failure (snapshot and catch-up completed)

before the next failure hit. In reality, the TMR system can tolerate a failure during the catch-up period, because after a recovering node successfully completes a snapshot request, it does not have to rely on the other nodes.

Evaluating performance during a joint failure of this type is more complex because the failures interact in a subtle way. For example, suppose that node number 1, recovering from a failure, has just completed its snapshot installation and is beginning to process a backlog of 10 transactions. Now say that node 2 detects a failure that happened during the execution of the first transaction after the node 1 snapshot transaction. Naturally, node 2 cannot detect that failure until the corresponding signature from node 1 is received. When node 1 starts its catch-up, it sends the required signature and node 2 then requests its snapshot. Unfortunately, node 1 will not get to this snapshot request until after it completes its own catch-up. Hence, the recovery of node 2 will take longer and more transactions than usual will be queued up. This may in turn delay a third recovery even longer, and so on. Since these joint failures are harder to study, and since they occur much less frequently, we did not study them in our experiments.

As mentioned earlier, we must control the ways in which a failed node can adversely affect the operational nodes. During a snapshot request, we cannot control how many pages are exchanged, but we can limit the transaction input rate. This in turn can limit the response time deterioration that transactions will suffer during this period. For example, if we expect exchange sizes of 20 or more and we would like to have an average response time of less than 500 ms, then Figure 4 tells us that the maximum input rate during recovery should be about 10 transactions per second.

Even if we can tolerate any increase in response time, it is necessary to limit the input rate to give a recovering node enough spare capacity to catch-up. As the simple empirical formula of the previous section indicates, if there is no spare capacity ($M-I = 0$), then the catch-up time will be unbounded. A long catch-up increases the chances of back to back recoveries as described above, which in turn delay the full recovery of the system and make it less reliable.

The experimental results can be useful in the selection of the appropriate input rate. For example, suppose that we wish to limit the total recovery time for a 50 page exchange size to 4 seconds. Since the exchange itself takes about 3 seconds (Figure 5), then we have to limit the catch-up to 1 second, or about 7 transactions per second (Figure 7).

5. CONCLUSIONS

In component-level TMR systems the recovery problem is simple because there is almost no state to recover. However, when TMR is used at the application level, recovery of the state, or database, is a significant problem. In this paper we have shown how a recovering node can obtain a valid copy of the database without halting the operational nodes. As we have seen, the process consists of a snapshot transaction that copies the corrupted parts of the database, followed by a catch-up process.

Our performance results are obviously not comprehensive since they are for a particular type of computer and transaction profile. However, since they are based on a real implementation of the algorithms, we believe they are illustrative of the types of loads and delays that will be encountered in recovery processing. They show that an application level TMR system does *not* mask out failures in the same sense that a component level one. Specifically, in an application level TMR system the recovery period will be "visible" to the users because of the reduced transaction input rates. This will have to be weighted against the advantages of application level TMR discussed in the introduction.

We also believe that our performance results illustrate the methodology required to evaluate TMR recovery. The results yield useful information for limiting the transaction input rate. They recovery and catch-up times they give can also be used in a probabilistic model to determine precisely the reliability of the system.

References

[Adib80] Adiba, M., and Lindsay, B., Database Snapshots, *Proceedings 6th International Conference on Very Large Data Bases*, October 1980, pp. 86-91.

[Cris84] Cristian, F., and Strong, R., Atomic broadcast: from Simple message diffusion to Byzantine Agreement, Research Report RJ-4540, IBM Research Laboratories, December 1984.

[Dole82] Dolev, D., and Strong, S., Polynomial Algorithms for Multiple Processor Agreement, *Proc. 14th ACM Symposium on Theory of Computing*, 1982, pp. 401-497.

[Dole84] Dolev, D., Halpern, J., Simons, B., and Strong, R., Fault-Tolerant Clock Synchronization, PODC Symposium, August 1984.

[Fisc83] Fischer, M. J., The Consensus Problem in Unreliable Distributed Systems (A Brief Survey), Technical Report YALEU/DCS/RR-273, Department of Computer Science, Yale University, June 1983.

[Fuch86] Fuchs, W., Wu, K., and Abraham, J., Low-Cost Comparison and Diagnosis of Large Remotely Located Files, *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, January 1986, pp. 67-73.

[Garc86] Garcia-Molina, H., Pittelli, F., and Davidson, S., Applications of Byzantine Agreement in Database Systems, *ACM Trans. on Database Systems*, Vol. 11, No. 1, March 1986, pp. 27-47.

[Lamp82] Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem, *ACM Trans. on Prog. Lang. and Systems*, Vol. 4, No. 3, July 1982, pp. 382-401.

[Lamp84a] Lamport, L., Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, *ACM Transactions on Programming Languages and Systems*, Vol. 6, Num. 2, April 1984, pp. 254-280.

[Lamp84b] Lamport, L., and Melliar-Smith, P., Byzantine Clock Synchronization, PODC Symposium, August 1984.

[Lind86] Lindsay, B., et al, A Snapshot Differential Refresh Algorithm, *Proc.*

SIGMOD Conference, May 1986, pp. 53-60.

[Lipt84] Lipton, R., Invariant Fingerprints, unpublished memo, Princeton University, 1984.

[Lund84] Lundelius, J., and Lynch, N., A New Fault Tolerant Algorithm for Clock Synchronization, PODC Symposium, August 1984.

[Lync82] Lynch, N., Fischer, M., and Fowler, R., A Simple and Efficient Byzantine Generals Algorithm, *Proc. 2nd Symposium on Reliability in Distributed Software and Database Systems*, 1982.

[Metz83] Metzner, J., A Parity Structure for Large Remotely Located Replicated Data Files, *IEEE Transactions on Computers*, Vol. C-32, No. 8, August 1983, pp. 727-730.

[Park83] Parker, D., et al, Detection of Mutual Inconsistency in Distributed Systems, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983, pp. 240-247.

[Peas80] Pease, M., Shostak, R., and Lamport, L., Reaching Agreement in the Presence of Faults, *Journal of the ACM*, Vol. 27, Num. 2, April 1980, pp. 228-234.

[Pitt86] Pittelli, F., Experimental Analysis of a Triple Modular Redundant Database System, Ph.D. Thesis, Princeton University, October, 1986.

[Schn82] Schneider, F., Synchronization in Distributed Programs, *ACM Trans. on Programming Languages and Systems*, Vol. 4, Num. 2, April 1982, pp. 125-148.

[Siew82] Siewiorek D., and Swarz, R., *The Theory and Practice of Reliable System Design*, Digital Press, 1982.

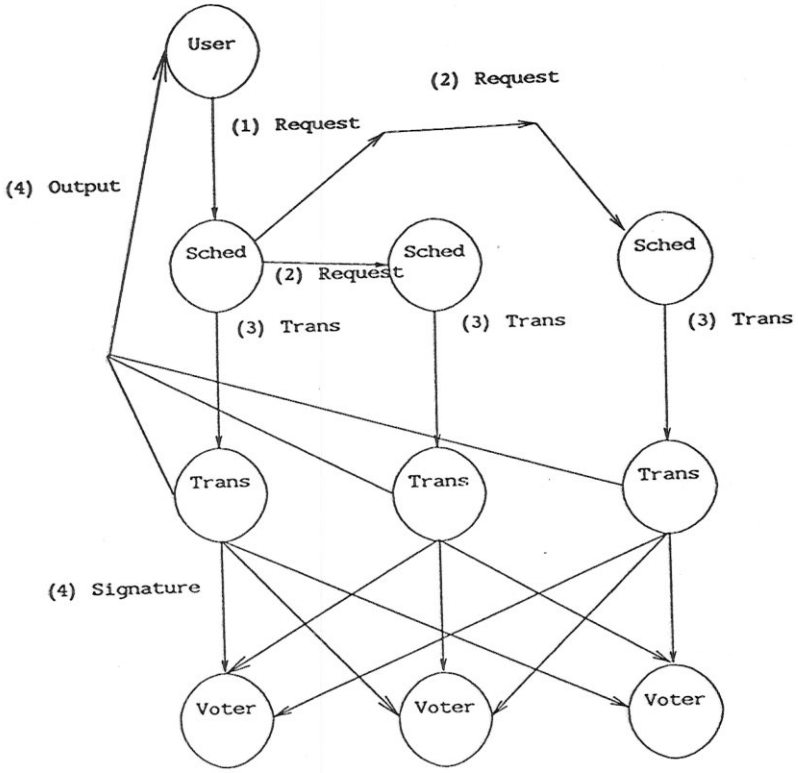

Figure 1. Normal Transaction Processing

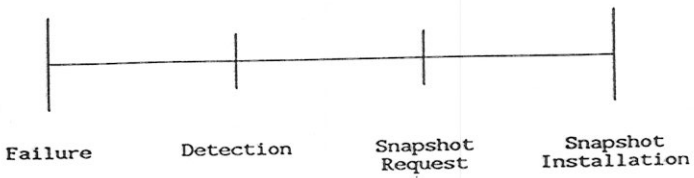

Figure 2. Recovery Model

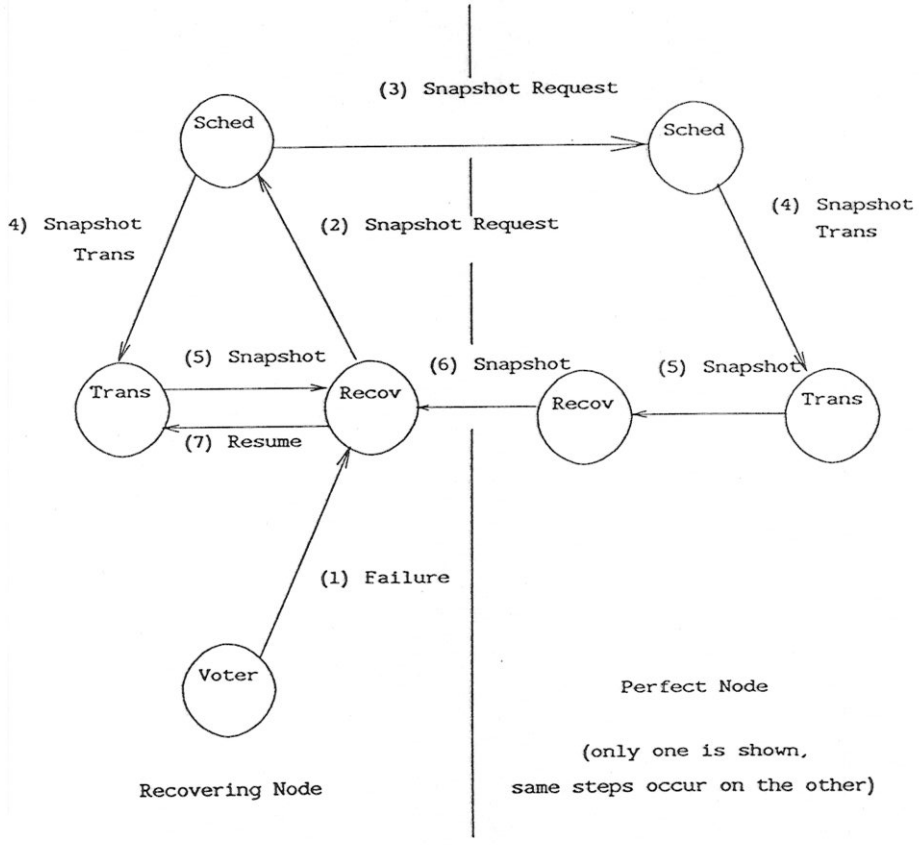

Figure 3. Recovery Processing

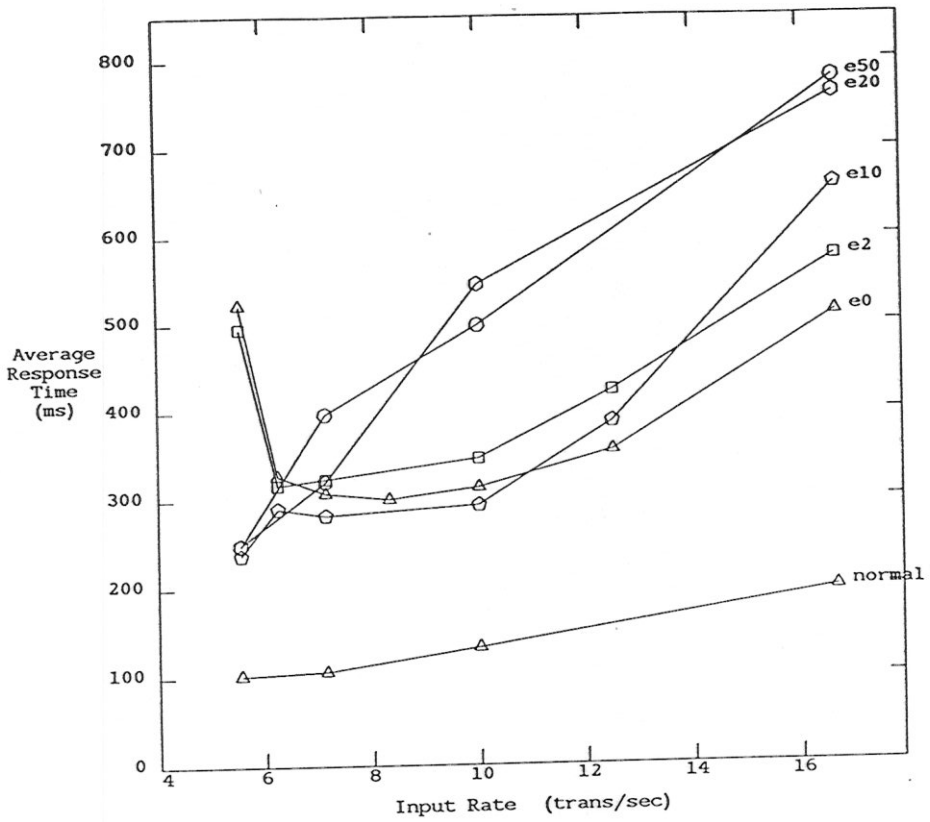

Figure 4. Impact of Exchange Size on Transaction Processing

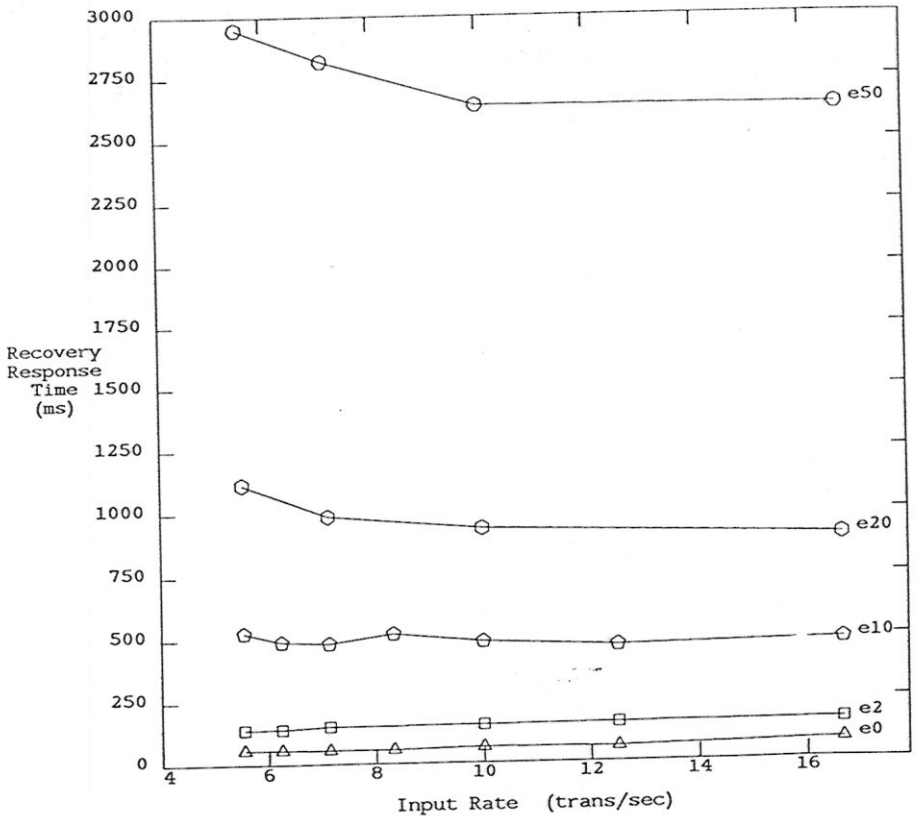

Figure 5. Recovery Response Time

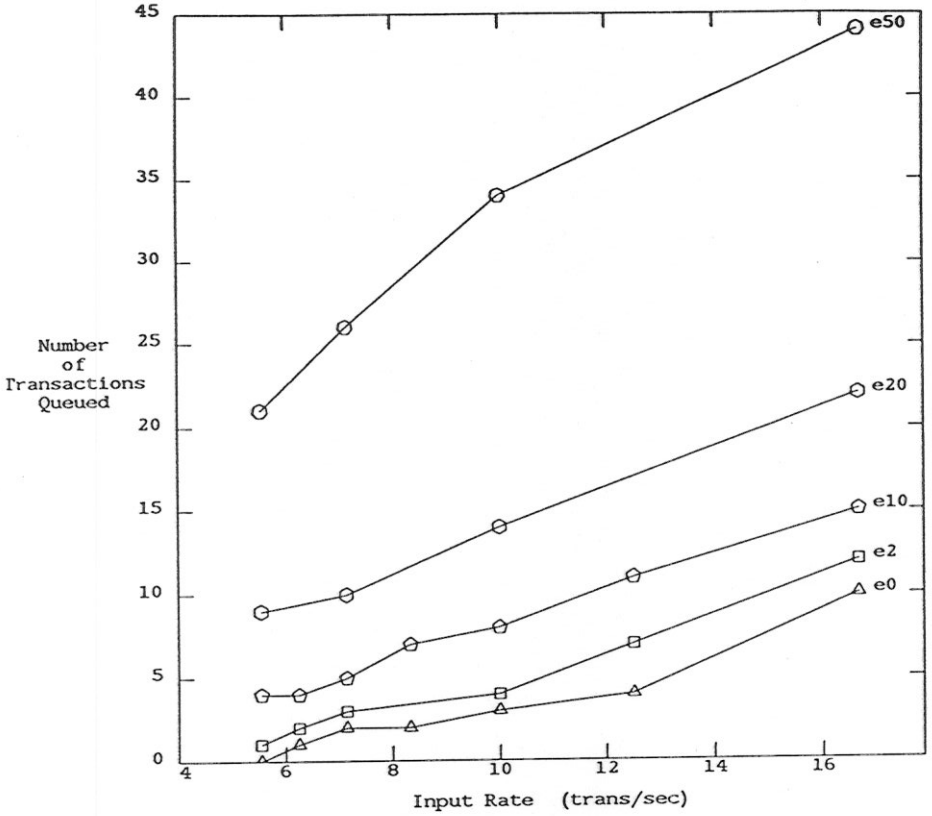

Figure 6. Impact of Exchange Size on Queue Size

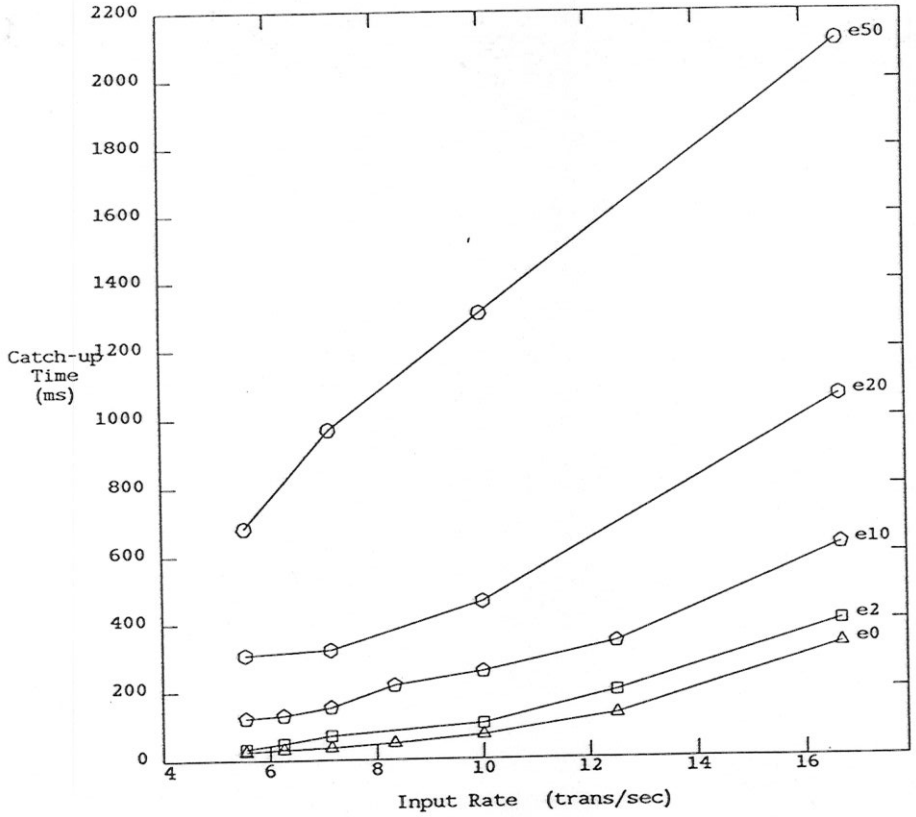

Figure 7. Impact of Exchange Size on Catch-up Time