DATA PROCESSING WITH TRIPLE MODULAR REDUNDANCY

Frank Pittelli
Hector Garcia-Molina

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

TR-002-85

Database Processing with Triple Modular Redundancy

Frank Pittelli
Hector Garcia-Molina

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

ABSTRACT

N-Modular Redundancy (NMR) protects against arbitrary types of hardware or software failures in a minority of system components, thereby yielding the highest degree of reliability. In this paper we study the application of NMR, specifically Triple Modular Redundancy (TMR), to general-purpose database processing. We discuss the structure and implementation tradeoffs of a TMR system that is "synchronized" at the transaction level. That is, complete transactions are distributed to all nodes, where they are processed independently, and only the majority output is accepted. We examine the inherent "cost" of such a TMR database system by presenting preliminary performance results from a version implemented on three SUN-2/120 workstations.

This work has been supported by NSF Grants ECS-8303146 and ECS-8351616, Naval Electronic Systems Command contract N00039-84-C0485, and from grants from IBM and NCR corporations.

June 27, 1985

Database Processing with Triple Modular Redundancy

Frank Pittelli
Hector Garcia-Molina

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

1. Introduction

In this paper we present an experimental database system that provides high reliability. It is designed as a collection of fully-replicated nodes that execute database transactions synchronously and will operate correctly even during periods of a single, "arbitrary" node failure. Our implementation experience and preliminary performance results have also shed light on the options available for achieving such reliability and the inherent costs associated with the approach. In this paper we discuss the design of the fully-replicated system, the implementation choices, and the performance results we have obtained.

For the most part, the organization of the paper follows the development of the system itself. In the rest of this section we present some important definitions and discuss our motivations. A description of the overall system is given in section 2, while sections 3, 4, and 5 discuss detailed implementation issues. Then, section 6 presents some preliminary performance results for several system configurations. Finally, in section 7, we make our concluding remarks.

1.1. Definitions

In order to design reliable systems we must define the types of failures that will be handled. This is usually accomplished by defining the "correct" operation of each system component in terms of assumptions about those components. In our discussions, we will make use of the following definitions.

Sane Failure

When a *sane* failure occurs a node "immediately" halts, without sending incorrect messages. When the node is restarted, it executes a recovery algorithm before resuming normal processing. (Such a node is called a *fail-stop* node in [Schl83].)

Insane Failure

An *insane* failure may cause arbitrary node behavior. (An insane failure is called a *malfunction* in [Peas80].) Such a failure can cause a node to send any message, including misleading ones, to other nodes. Furthermore, it can refuse to send required messages and can even collaborate with other insane nodes in an attempt to subvert the entire system.

Perfect Node

A *perfect* node is one that does not fail during a given period of time. This definition is used primarily when discussing the operation of distributed algorithms. For example, to mask a single insane failure, at least two perfect nodes must exist. This does not mean that the nodes never fail. It simply means that, during the duration of the insane failure, the nodes do not fail. In general, most distributed algorithms dealing with insane failures require some number of perfect nodes during certain critical periods of the algorithm.

Stable Storage

A node is considered to have *stable storage* if updates to that storage are resilient to sane failures [Lamps79]. That is, if a processor halts in the middle of a storage operation, either the operation is completely successful or it is not performed at all. Usually, stable storage is implemented using disks.

1.2. Motivation

Most database systems assume that only sane failures can occur. This assumption usually allows the development of efficient algorithms. Unfortunately, it may be unacceptable when database systems are used for critical applications (e.g., controlling a nuclear reactor) or when nodes fail in unpredictable ways. Consequently, systems designed to handle insane failures are being developed for database processing.

One simple method for coping with insane failures "replicates" all the work done by the system and "compares" the outputs to determine a single, "correct" system result. Usually, such systems contain three fully-replicated processing elements and are known as *Triple Modular Redundant* (TMR) systems [Siew82]. The main advantage of TMR systems is that no matter what caused the failure, or what it affected, the majority result is chosen as the system result, thereby making the system very reliable. Additionally, TMR techniques may be incorporated at many different levels within a system. For example, gate-level components, modules, or complete computer systems may be replicated. Finally, since comparisons are performed only on component outputs, it is possible to incorporate standard processing nodes into the TMR design.

TMR techniques have been used extensively in process control environments (e.g., airplane and spacecraft control), but their use has almost always involved simple inputs, outputs, and "system states". For example, the SIFT (Software Implemented Fault Tolerance) computer controls wing actuator settings based on various sensor readings [Wens78]. The input and outputs are voltages from hardware devices. On the other hand, we are interested in using TMR for database applications. That is, the input is an entire database transaction, while the output is a, possibly large, subset of the database. Furthermore, the system state must encompass all the data contained in the database itself. We believe that such high-level interaction can significantly reduce communication costs, allow us to use conventional database processing, and simplify the design of the reliable system. Such processing, however, introduces some interesting problems, especially with respect to failure detection and recovery, that have not been fully examined in the literature. In this

paper, we will outline these problems and some potential solutions.

One common criticism of TMR systems for database applications is that their cost, both in terms of resources and operating expenses, may be "prohibitive". Of course, the extra cost does provide added reliability, so both factors must be considered together. This paper provides a framework for such comparisons, allowing us to evaluate the TMR database techniques in a fair and objective fashion.

Along the same lines, TMR systems have been criticized because they are "fully-replicated". That is, some people believe it is impractical to replicate an entire database on three nodes. However, in many systems a small, but critical, portion of the database must be maintained with high reliability. In such a situation, the critical portion may be maintained by a TMR subsystem. (IBM's Highly Available System protects certain system tables using a similar approach [Aghi83].) Our experimental system concentrates on the operation of such fully-replicated, database subsystems.

Note that TMR techniques, including the ones presented in this paper, can be easily generalized to NMR techniques. However, we will restrict our discussion to triple redundancy, both because it is the most likely in practice and because our experimental results are for such a level of replication.

Finally, a detailed discussion of database processing in the presence of insane node failures can be found in [Garc85]. In that paper, models are presented for node and network behavior and a correctness criteria for the overall database system is defined. Throughout this paper, we will be discussing the implementation issues associated with such a replicated system. Readers interested in the "theory" behind the implementation should refer to the paper cited.

2. System Design

An important advantage of a TMR system is its simple design. All nodes perform the same functions and contain the same data. Furthermore, it is possible to design a node's operation around separate components, each responsible for a distinct process. For example, one

component on each node may provide for the synchronization of transaction execution, while another performs the transactions themselves. In our system, each node consists of the following components.

Scheduler
Transaction Manager
Voter
Recovery Manager

Also, user processes, located throughout the processing nodes or on other hosts, submit transactions to be performed and wait for transaction results to be returned. In order to describe the operation of these components, we will trace a single transaction as it is processed by the system (Figure 1).

2.1. Scheduling

A user process begins the execution of a transaction by submitting a transaction request to one of the schedulers. It is the scheduler's task to guarantee that all nodes in the system receive the transaction request, even in the presence of node failures. Additionally, the schedulers must guarantee the same "transaction commit order" (i.e., the system schedule) on all nodes. In short, the schedulers make use of synchronized clocks, a reliable broadcast algorithm and "synchronization" messages to maintain identical schedules on all perfect nodes. We will discuss some of the scheduling algorithm details in section 3.

2.2. Transaction Processing

Once a transaction has been scheduled by the local scheduler, it is sent to the local transaction manager. The transaction manager, in turn, executes them against the database as fast as possible. The actual processing of transactions may be implemented in many different ways. (We will discuss various options in section 4.) However, in all cases, the transaction manager must "commit" the transactions in the order they were scheduled.

Once a transaction has been completed, the output of the transaction is sent to the user, while the current "signature" of the database is sent to each voter (see below). The user determines the "correct" output of the transaction by selecting the output produced by a majority of nodes. If no correct output can be determined, the user may submit a special transaction to discover the outcome of the questionable transaction.

2.3. Failure Detection

The database signature of each node is used to determine when a node failure has occurred. That is, a given node can determine that "it" has failed by comparing its database signature to that of the other nodes. If the signature received from the local transaction manager disagrees with that from a majority of the transaction managers, the voter assumes that some local component has failed and sends a failure report to the recovery manager. (The voter can only detect the failure after the node has been repaired and is functioning properly. Once it is repaired, the node executes its algorithms correctly, but the database contains incorrect data.) This form of failure detection allows the voter to detect transient processing errors, lost transactions, processor halting, and spontaneous media failures. For the most part, the voter doesn't care *what* failure happened, only that *some* failure caused the database to be inconsistent.

The signature used by the voters must encode the effects of each transaction on the database and should be relatively easy to compute and maintain. Such a signature may be completely deterministic (e.g., a redo log) or it may be probabilistic (e.g., a "checksum").

2.4. Failure Recovery

Once a possible node failure has been detected, the recovery manager is responsible for bringing the local database back to a consistent state. Of course, since transactions may have been processed by the other nodes during the failure, the recovery manager must depend on the other nodes in the system to determine the correct database state.

Once again, we will present the basic recovery procedure by following the flow of messages through the system. Keep in mind, the primary goal of the recovery protocol is to allow a failed node to restore its database, without noticeably interrupting transaction processing at the other nodes.

To start the recovery process, the voter informs the local recovery manager of a failure (Step 1 in Figure 2). 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 perfect nodes (Steps 3 and 4). (Note, if the recovering node can't participate in this snapshot transaction 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 sends it to the local recovery manager and resumes normal transaction processing (Step 5). (Actually, the transaction manager on the recovering node must postpone processing. We will discuss this shortly.) At this point, the recovery managers send their snapshots to the failed recovery manager, who uses them to correct the local database. Finally, the failed transaction manager is given a correct copy of the database and may resume transaction processing (Steps 6 and 7).

There are many different protocols that may be used by the recovery managers to send their snapshots to the recovering node. If the database is small, the entire snapshot may be transmitted to the other node. Alternately, a database "signature" may be used to detect the database pages that have been corrupted and need to be transmitted to the failed node. (This signature need not be the same one used by the voters to detect a failure.) However, in all these cases one point remains the same. It always takes some finite amount of time to install the consistent copy after the snapshots have been taken. Therefore, transaction requests that arrive during this time and are processed by the normal nodes, must

be stored by the recovering node and processed at a later time. Consequently, there is some time period, possibly infinite, when the recovering node is trying to catch-up to the other nodes in the system. (Actually, transactions may be executed before the database is consistent, but this may cause another failure. Pay your money and take a chance !)

At this point, we discuss the implementation details associated with some of the system components.

3. Scheduler

The schedulers must guarantee the following properties in the presence of node failures.

1) If any perfect scheduler schedules a transaction, then all perfect schedulers schedule that transaction. Furthermore, if a faulty scheduler schedules a transaction, then either all or none of the perfect schedulers schedule that transaction. Note, we do not require that *all* transactions be scheduled.

2) If any perfect scheduler schedules a set of transactions in a given order, then all perfect schedulers will "eventually" schedule that set of transactions in the same order. In particular, we do not require that transactions be scheduled simultaneously by all perfect nodes, only eventually.

The first property is a version of the Byzantine Generals Problem [Lamp82] and has many proposed solutions [Dole82, Lync82, Peas80]. (Any algorithm which solves this problem is called a "reliable broadcast" algorithm.) The second property essentially requires that all "actions" on the global schedule be "synchronized". Two approaches have been proposed to guarantee such a property. Lamport has described a "state machine approach" which makes use of reliable broadcasts and synchronized real clocks to cope with arbitrary node failures [Lamp84]. To the same end, Schneider makes use of reliable broadcasts and logical clocks, implemented via acknowledgement messages [Schn82]. Both approaches are very general, and can be applied to many different distributed applications. Consequently, the design of the scheduler incorporates ideas from both. (In fact, the entire database system presented

in this paper can be considered a manifestation of these approaches.)

A complete presentation of the scheduler design is beyond the scope of this paper. Instead, we will discuss two important implementation issues. (Readers interested in a detailed discussion of the scheduler design and operation should refer to [Pitt85].)

3.1. Null Transactions

As mentioned, Lamport's state machine approach makes use of synchronized clocks to guarantee a unique ordering of transactions by all perfect schedulers. Essentially, each transaction is assigned a globally unique timestamp that is used to correctly schedule it, despite an insane failure. It can be shown that the timestamp must depend on the "maximum" message delay imposed by the system [Lamp84]. Unfortunately, most messages are processed far more quickly. For example, in our implementation the maximum message delay is about 500 ms, while the average message delay is about 11 ms.[1] Consequently, it is advantageous to "prematurely" schedule transactions when no failures occur.

A given scheduler can determine that a transaction can be prematurely scheduled by monitoring the flow of transactions from each scheduler. For the most part, if all schedulers have transactions pending, then the transaction with the smallest timestamp can be scheduled immediately [Schn82]. That is, the transaction can be scheduled *before* its timestamp expires. In this way, the scheduling delay associated with each transaction is independent of the maximum message delay, given that there are no failures and at least one transaction is pending from each scheduler. (The scheduling delay is the amount of time it takes to schedule a transaction.)

During slack periods, a steady supply of pending transactions can be provided by "null" transactions [Lamp84, Schn82]. Essentially, schedulers submit null transactions when they have no real transactions pending and at least one transaction is awaiting premature scheduling.

---

1. The large difference is caused by the operating system, not the network hardware. The details of our implementation, e.g., hardware used, are given in section 6.

Null transactions are handled like any other transaction, with the exception that they are not sent to the transaction manager when scheduled.

There is, of course, a tradeoff associated with the use of null transactions. When few user transactions are being processed concurrently, null transactions allow premature scheduling and this decreases the average scheduling delay. On the other hand, when many user transactions are pending, premature scheduling can be accomplished without the use of null transactions, which cause additional processing. Consequently, the "policy" used to generate nulls must take both factors into account. To illustrate this point, Table 1 shows the number of null transactions generated by the currently-implemented policy, as a percentage of the total number of transactions processed, for varying degrees of multiprogramming. Note, as multiprogramming increases fewer null transactions are generated. Conversely, with low concurrency null transactions account for most of the processing done by the schedulers, but the system is lightly loaded anyway. (The details of the current null policy can be found in [Pitt85].)

Table 1. Null Transaction Processing

Multiprogramming	% of Null Trans
1	67
2	61
3	55
4	50
5	42
6	39

It is interesting to note that most of the scheduling delay associated with a transaction is due to "synchronization", rather than "distribution". That is, a transaction itself is distributed to all nodes through one reliable broadcast. Such a broadcast may or may not be "expensive", depending on the failure model used and implementation details.

However, a transaction can't be scheduled prematurely until the other schedulers have issued transactions (user or null) with later timestamps, and these require additional broadcasts.

3.2. Batching Transactions

The reliable broadcasts used by the schedulers produce many messages for each transaction. In particular, in a three node system four messages are generated for each transaction (real or null) submitted by a scheduler (Figure 3). A decrease in message processing may be achieved by scheduling transactions in "batches". That is, each scheduler collects a number of transactions before submitting them to the other schedulers, where the entire batch is scheduled at one time. Of course, this strategy has both advantages and disadvantages.

On one hand, transaction batching decreases the total number of messages generated. For example, if three transactions are batched together, then eight messages are saved during the scheduling process (recall Figure 3). Of course, this decrease in the number of messages also decreases the amount of message "processing" performed by the schedulers. (Message processing includes the transmission time and the CPU time used by the sender and receiver.) Figure 4 shows the average processing time for messages of varying sizes in our system. We can see that the system-imposed overhead for each message (i.e., the y-intercept) is about 8.6 ms. Therefore, by batching N transactions together we may be able to save $4 \times 8.6 \times (N-1)$ ms of processing time, thereby increasing system throughput.

On the other hand, the schedulers must know "when" to submit the current batch of transactions. For example, with two transactions waiting in the batch it may be 2 ms or 2 days before a third arrives. Consequently, the scheduler should only hold a batch until it has been waiting a certain period of time or until it has been filled (i.e., the largest possible message size has been reached). Furthermore, batching may increase the average scheduling delay because transactions are sometimes "waiting" for a batch to be released.

4. Transaction Manager

The transaction manager may be implemented using many different conventional database techniques. This flexibility is due to the small number of requirements imposed by the system design. In fact, the transaction manager is only required to commit transactions in timestamp order and to maintain a database snapshot. Now we discuss a few of the choices.

4.1. Stable Storage

Unlike normal database processing systems, the transaction manager does not need to use stable storage for the database. If the database is lost during a failure, the node can recover by using the data from the other nodes. However, it may still be advantageous to keep the database in stable storage. If a sane failure occurs and the recovering node is able to recognize this, then it can use the local copy when recovering, thereby decreasing the recovery period. That is, at recovery time the local database is consistent and the node need only acquire the changes performed while it was down. Of course, even with stable storage, an insane failure may destroy the database, making use of the remote copies necessary.

Stable storage may help at recovery time, but there is a cost associated with its use. That is, a transaction manager that does not use stable storage should be able to maintain a higher processing rate, during no-failure periods, than one that does. This increase in performance may or may not compensate for the increase in recovery processing. The best choice will depend on the frequencies of each type of failure and on the overhead of each storage strategy.

In the current implementation, the database consists of a collection of pages maintained on disk. Additionally, a cache of recently-used pages is maintained in memory. Any transaction requiring a given page may find it in the cache or may read it from the disk. Any update to the page is performed on the cache copy. If no stable storage is desired, updated cache pages are written to the disk only when a cache slot is required for another page. Consequently, a node failure may cause previous updates

to be lost.

If stable storage is desired, two extra steps are taken. First, as a transaction modifies cache pages, they are "pinned" in memory. That is, the cache pages are not allowed to overwrite the corresponding disk pages until the transaction commits. (At commit time, the pages are flushed to disk.) Given a sufficiently large cache, pinning is not a restrictive operation. Secondly, to survive a sane failure during the flush period, a "redo" log is maintained. That is, before any page is written to the disk the updated data within that page is written to a sequential log. Once all log records have been written, a commit record is written on the log and the transaction output is sent to the user. At this point, the updated pages may migrate to the disk at any rate. When all the pages for a transaction have migrated the corresponding log records are deleted. (This keeps the log size manageable.)

4.2. Degree of Multiprogramming

The simplest transaction manager is implemented by a single process. This process receives transactions from the scheduler and executes them serially. No concurrency mechanism is needed, whether stable storage is implemented, or not. Such a transaction manager is easy to implement and has been examined first in our experiments.

A performance gain may be achieved by creating multiple transaction servers. In this way, the transaction manager receives transactions from the scheduler and assigns the transaction to a server. Each server is capable of performing a single transaction at a time, similar to the simple transaction manager described above. Of course, with multiple servers, some concurrency control mechanism must be used to guarantee that the transactions are performed in the correct sequence. In fact, many "standard" concurrency control algorithms may be adapted to ensure this. For example, two-phase locking can be modified so that a transaction which requires a lock held by another transaction, with greater timestamp, can simply abort the later transaction and take the lock. Along these lines, we are implementing a multi-server transaction manager based on the PREDATOR Database System [Kent85].

4.3. Snapshots

As stated in section 2, the transaction manager is responsible for taking a snapshot of the database. This snapshot is used by the recovery manager when a node is recovering from a failure. Of course, during the recovery phase of another node, the local transaction manager must continue normal processing without disturbing the timestamped snapshot.

In our system, the transaction timestamps are used to simplify snapshot generation. In particular, every page in the database is marked with its most recent update time. Additionally, each page may have two different disk copies. At any time, every page has a copy that has a timestamp less than or equal to the current snapshot time. This copy is called the "snapshot" copy. If only the snapshot copy exists, it is copied to a free page before any transaction may update that page. This new copy will have an update time greater than the snapshot time. If two copies exist and both have update times earlier than the snapshot time, then the oldest copy is deleted and the previous condition is applied. Finally, if a copy exists with an update time greater than the snapshot time, then it may be updated directly by the transaction.

This strategy provides efficient snapshot generation. A new snapshot is generated simply by advancing the snapshot time. Any page updated after the snapshot will be copied automatically.

4.4. Database Signature

A database signature may be exchanged at two different times. First, whenever a transaction is performed, the resulting database signature is sent to each voter. The voters use this signature to determine if a failure has occurred. Since the voter receives a signature after every transaction is executed, the signature should be easy to compute and transmit. Additionally, the voter need not determine the pages that have been corrupted. Rather, the voter simply has to detect the presence of a corrupted page. Currently, the signature used by the voter is represented by a four-byte checksum. The checksum is computed from the checksums of all the pages accessed by a given transaction. In this way, corrupted pages can be detected with high probability. (In practice, the

checksum for an entire page is computed once when the database system is started. Thereafter, each checksum is "adjusted" when the corresponding page is modified, providing efficient maintenance.)

Whenever a recovery is in progress, the recovery manager may use a database signature to determine the pages that have been corrupted and, consequently, must be transferred from the other nodes. Unlike the voter signature, this database signature is rarely transmitted by the transaction manager. Furthermore, this database signature must "pinpoint" the pages that have been corrupted. Depending on the size of the database, three types of database signatures will be examined.

If the database is small, as in the current implementation, the database signature can be represented by an array of checksums. There is one checksum for each page in the database. The entire checksum array is exchanged by the recovery managers as required. Additionally, the entire checksum array can be maintained in memory, even for medium size databases, thereby making it computationally efficient.

For large databases, it is impractical to transfer an array of checksums. Rather, a "hierarchy" of checksums is maintained. That is, all database pages are grouped into segments, which, in turn, are grouped into larger segments. A checksum for each segment is computed based on the checksums for the pages (segments) within that segment. Using this tree of checksums, the recovery managers can iteratively determine the pages in the database that have been corrupted. For example, consider a database with 10,000 pages organized into 100 segments of 100 pages each. In the first phase of a recovery, the recovery managers exchange only the checksums of the 100 segments. For any segment that is corrupted (as determined by its checksum), the checksums of its 100 pages are exchanged. If transactions access pages within a single segment, only 200 checksums will be exchanged to recover from a single transaction failure. Clearly, the size of the database segments, as well as the transaction access patterns, determine the efficiency of such a strategy.

Finally, for large databases, the complete signature must be easy to compute. In particular, a recovering node may have to reconstruct the

entire database signature before initiating the restart protocol. Such a situation may be helped by a probabilistic signature proposed by Lipton. [Lipt84] His strategy produces a signature that can be used to detect, with high probability, differences between two databases. More importantly, the signature can be computed using any ordering of the database pages. In this way, a recovering node can compute the signature by sequentially accessing the "physical" database pages, thereby making its computation efficient.

5. Failure Detection

As previously described, the voter is responsible for detecting the failure of the local node, by comparing all outputs from all nodes. The implementation of such a voter should consider the following details.

5.1. Vote Queue

Since the three nodes will probably be executing transactions at different rates, it is advantageous to maintain a queue of pending transactions. When all signatures for a given transaction have been received, they are compared. Additionally, if the local signature agrees with another (thereby forming a majority), before the third is received, the voter may delete the transaction from the queue. The other signature will simply be ignored when it arrives. This strategy helps to decrease the average size of the voter's queue. Of course, the voter must always wait for the local signature to arrive before it can delete a transaction from the queue.

5.2. Time Limit

If the local signature for a transaction is lost (or the transaction was not performed by the local node) the voter must decide when to report the "failure" to the recovery manager. That is, after seeing a majority from the other nodes, the voter should only wait some finite amount of time before informing the recovery manager. Unfortunately, choosing a time limit is not easy. For example, if the local transaction manager is heavily loaded (e.g., when trying to catch-up after a recovery), the

signature for each transaction will arrive much later than those for the other nodes. Therefore, if the time limit is too small, the voter will report a "failure" even though the transaction was performed correctly. (The current recovery protocol isn't affected by such a situation, but the imposed overhead is unnecessary. Additionally, a real failure on another node can't be handled during this period.)

The easiest way to remedy this situation is to select a large time limit. Unfortunately, preliminary research shows that the length of the time limit directly determines the number of pages that must be exchanged during a given recovery cycle. (The longer a failure goes unreported, the greater the probability that other transactions will access the corrupted pages and corrupt other pages.) Therefore, we must select a time limit that is large enough to prevent false failure reports, yet small enough to reduce the amount of recovery processing.

5.3. Vote Batching

There is a method that may be used to eliminate the effects of a local signature message that is lost. If the signature reflects all pages in the database, not just those accessed by a single transaction, it may be used to delete a group of transactions from the queue. For example, the currently used signature is generated from the checksums for *all* of the database pages. If a signature message is lost, but the pages are correctly updated, the next signature message will reflect the changes. Therefore, for a given transaction, if the voter sees that the local signature agrees with the majority, it can delete all pending transactions that have a timestamp less than the current one. Conversely, if the local signature disagrees with the majority, all pending transactions can be deleted.

Such a modification can be extended to gain another advantage. Similar to batch scheduling, the transaction manager could decide to send a signature for a batch of transactions. However, in this case, instead of combining a group of signature messages, all but the last are eliminated. This should improve the load on the network and the overall system performance, without drastically affecting the recovery

characteristics.

6. System Performance

In this section, we present preliminary performance results for a triple node database system. In particular, we examine the average response time and the system throughput, for periods of normal processing. Of course, since our system is experimental, "absolute" values for the transaction response time and the system throughput may be misleading. It is more advantageous to compare the performance of the triple node system to other, less reliable systems, implemented using the same basic design. In this way, we can begin to determine the performance costs associated with an increase in reliability. To this end, the basic system design has been used to implement a single node and a dual node database system.

Unless otherwise stated, each system makes use of the processing components described in the preceding sections. Specifically, the schedulers are designed to make use of null transactions to reduce the scheduling delay, but batch scheduling is not used. Additionally, the transaction manager processes transactions serially, *without* the use of stable storage, while the voters receive signatures for each transaction. Finally, the recovery managers make use of an array of checksums, as a database signature, to determine the corrupted database pages.

6.1. Configurations

Each of the database systems were implemented on a collection of SUN-2/120 workstations, running SUN UNIX (based on BSD 4.2 UNIX), connected by a 10-Mbit/sec ethernet.[2] 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

2. We realize that the ethernet is a single point of failure in our experimental system, but we are using it for a practical reason: we have no other. However, our design does not require the "broadcast" capability of the ethernet. Any direct connection between the processing nodes will suffice. Furthermore, we believe that a different network will not change our performance results significantly.

SUN UNIX. Finally, all transactions originated from a single user process located on a VAX-11/750, running BSD 4.2 UNIX.

Now, we briefly describe the configuration of each system.

Single Node System

The single node system consists of one transaction manager. Transactions are sent directly to the transaction manager from the user. Since only one node exists (and no stable storage is used), there is no recovery from failure, and therefore, no recovery manager. Similarly, no voter or scheduler is needed. Note, since the single node system doesn't use stable storage, it doesn't represent a *conventional* database system. Rather, it represents an "upper bound" against which to compare. That is, it yields the best possible performance by neglecting reliability considerations.

Dual Node System

The dual node system consists of a scheduler, a transaction manager and a recovery manager on two nodes. It is capable of recovering from a single, *sane* failure of either node and may execute transactions during such a recovery, although at a reduced rate. Finally, the transaction managers have been modified to detect their own failures. (This is possible because of the nature of sane failures.)

Triple Node System

The triple node system consists of all components on three nodes, where voters are used to detect node failures. The system is capable of recovering from a single, *insane* node failure and allows transaction processing during recovery periods.

6.2. Database Transactions

The database used in our preliminary experiments consists of 100, 512-byte pages, while the database cache (in memory) can hold a total of 16 pages. Furthermore, each database transaction updates a fixed number of pages chosen randomly with uniform probability, where each

update affects a single byte chosen at random. This combination of a small database, a relatively large cache, and a simple transaction allows efficient processing.

It should be noted that these parameters would normally be considered unrealistic. However, such parameters tend to exaggerate the impact of message processing on the overall system performance. That is, since transactions are processed quickly, a comparable portion of the system performance is devoted to message processing. Such an exaggeration allows use to focus on the change in message processing as the number of nodes is increased, and that is the goal of our preliminary experiments. Future experiments, involving recovery processing and more complicated transaction managers, will make use of more realistic transactions and databases.

6.3. Response Time

In each system, the average transaction response time was determined when a *single* user existed. That is, there was only one transaction being processed by the system at any given time. Such a situation yields results that are free from the effects of multiprogramming. In the single and dual node systems the response time was taken to be the time between sending a transaction request and receiving the *first* transaction output. In the triple node system it was taken to be the time until two identical outputs were received. In all cases, the average was determined based on a sample set large enough to insure an error of less than five percent.

Figure 5 shows the average response times for each of the systems. In all three curves, the slope is about 11 ms/page. That is, each database page access (a read and a write) takes about 11 ms. Additionally, the y-intercepts for each curve are 16, 76, and 130 ms. These values represent the response time for a transaction that accesses no database pages. The differences in these response times can be explained as follows. (Recall, the average message delay is 11 ms.)

In the dual node system at least three extra messages are exchanged, one to broadcast the transaction request to the other

scheduler, one to send a "null" transaction back to the input scheduler, and one to send the transaction to the transaction manager. Additionally, since node clocks may differ, the scheduler may wait some amount of time before sending the null transaction. (In our implementation, the clocks differ by about 20 ms.)

In the triple node system most of the extra 114 ms is attributable to the reliable broadcast (2 message delays), the synchronization (i.e., null) messages (4 message delays) and the clock differences (20 ms). The remaining time delay is a result of the user waiting for two outputs instead of one.

6.4. System Throughput

The average transaction response time is useful for determining the additional message delays imposed by a system, but it doesn't accurately reflect the amount of "parallelism" within the system. For example, with an infinite capacity processor, the scheduling of one transaction can be done concurrently with the database processing of another. Additionally, multiple transactions can be scheduled by the schedulers concurrently. (If designed properly, the transaction manager can also execute transactions concurrently, but the current transaction managers do not provide this flexibility.) With a less powerful processor, however, these concurrent operations degrade the performance of each other. In order to determine this degradation, the system throughput must be examined.

For each system, the system throughput was determined by saturating the transaction manager at each node.[3] That is, for each system and transaction size, a level of multiprogramming was chosen which kept the transaction manager busy and maximized the system throughput. Figure 6 shows the "best" throughput values for each system and transaction size. Note, for each system the throughput increases as the transaction size increases. Essentially, since the scheduling delay is a "per transaction" overhead, the larger transactions are executed more efficiently.

---

3. Much to our dismay, measuring system throughput during saturation periods is difficult. The strain on the system, due to the processing load and the large number of messages, produces a large variation in results.

Using the single node system as a base, we can see that the dual node system has a throughput of 60 to 76 percent. Clearly, the degradation is a result of the concurrent scheduling of transactions. Furthermore, the curves suggest that larger transactions experience a proportionally smaller degradation.

Similarly, the triple node system shows a throughput of 26 to 54 percent. Part of the throughput degradation is attributable to the increase in synchronization messages, while another part is a result of voter processing. Recall, in the dual node system the transaction manager detects failures, while in the triple node system this role is performed by the voter processes. Although the actual comparison of transaction outputs is computationally inexpensive, many signature messages (9 per transaction) must be processed by the voter and operating system on each node, and this decreases system performance. (Remember, no batching (scheduling or voting) was implemented in the current experiments.)

Finally, the reader should keep in mind that the systems were implemented on micro-computer based workstations. Consequently, the system degradation imposed by concurrent processes is exaggerated. Also, the transaction manager has been implemented without stable storage, using a small database. Due to such simplicity, message processing is comparable to, if not greater than, database processing. Presumably, database systems that make use of stable storage and a larger database will show a proportionally smaller throughput degradation as the number of nodes is increased. (Future experiments will attempt to quantify this statement.)

7. Conclusion

In this paper we have presented a reliable database system designed using TMR techniques. The system consists of three fully-replicated database nodes that execute transactions synchronously. Each node consists of a collection of components, each of which performs a well-defined part of the overall system task.

Our preliminary results show that the average scheduling delay is relatively independent of the transaction size. Consequently, the system

achieves its greatest throughput when processing large transactions. That is, as transactions require more processing, the scheduling overhead imposed by the TMR design decreases proportionally.

An important aspect of the system design is that transactions are performed by the transaction managers without inter-node communication (e.g., no remote lock request and no distributed deadlock detection). Consequently, the transaction manager can be implemented using conventional database techniques. Also, since the database is replicated at three nodes, there is no *need* to use stable storage. Without stable storage, the transaction managers may be able to sustain high processing rates.

Failure detection is always a difficult problem and has been solved in our TMR design by using a database signature. The signature provides an efficient method to detect incorrect transaction outputs. However, the number of messages required for failure detection in a TMR system may cause a degradation in system throughput. Once a failure has been detected, the recovery manager may use another type of signature to determine the corrupted database pages. Then, the correct database pages are acquired from the other recovery managers.

In our TMR database system, the main performance costs are due to synchronization and failure detection messages. Therefore, we have considered two methods to reduce the number of messages. First of all, transaction batching may be used during the scheduling phase. Similarly, signature messages, used for failure detection, may be transmitted for a group of transactions, instead of singly. In both cases, the decrease in message traffic should allow greater system throughput, without significantly affecting other system characteristics.

Finally, our experimental results allow us to quantify the cost of a TMR database system. In particular, we can examine the scheduling delay of each transaction. In other words, we can begin to understand the amount of processing required to "synchronize" actions in a fully-replicated distributed system. Secondly, our design allows us to investigate the performance of systems with varying degrees of reliability and availability. Future experiments will pursue these topics in more detail.

References

[Aghi83] Aghili, H., et al, A Prototype for a Highly Available Database System, Research Report RJ-3755, IBM Research Laboratories, January, 1983.

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

[Garc85] Garcia-Molina, H., Pittelli, F., and Davidson, S., Applications of Byzantine Agreement in Database Systems, to appear in *ACM Trans. on Database Systems*, 1985.

[Kent85] Kent, J., Performance and Implementation Issues in Database Crash Recovery, Ph.D. Thesis, Princeton University, June, 1985.

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

[Lamp84] Lamport, L., Using Time Instead of Timeout for Fault-Tolerant Distributed Systems, *ACM Transactions on Programming Languages and Systems*, Vol. 6, Num. 2, pp. 254-280, April, 1984.

[Lamps79] Lampson, B., and Sturgis, H., Crash Recovery in a Distributed Data Storage System, Xerox Research Memo, April, 1979.

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

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

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

[Pitt85] Pittelli, F., Performance Analysis of Transaction Scheduling in a TMR Database System, in preparation, 1985.

[Schl83] Schlichting, R.D., and Schneider, F.B., Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems, *ACM Transactions on Computer Systems*, Vol. 1, Num. 3, pp. 222-238, August, 1983.

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

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

[Wens78] Wensley, John, et. al., SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control, *Proc. of IEEE*, Vol. 66, pp. 1240-1254, October, 1978.

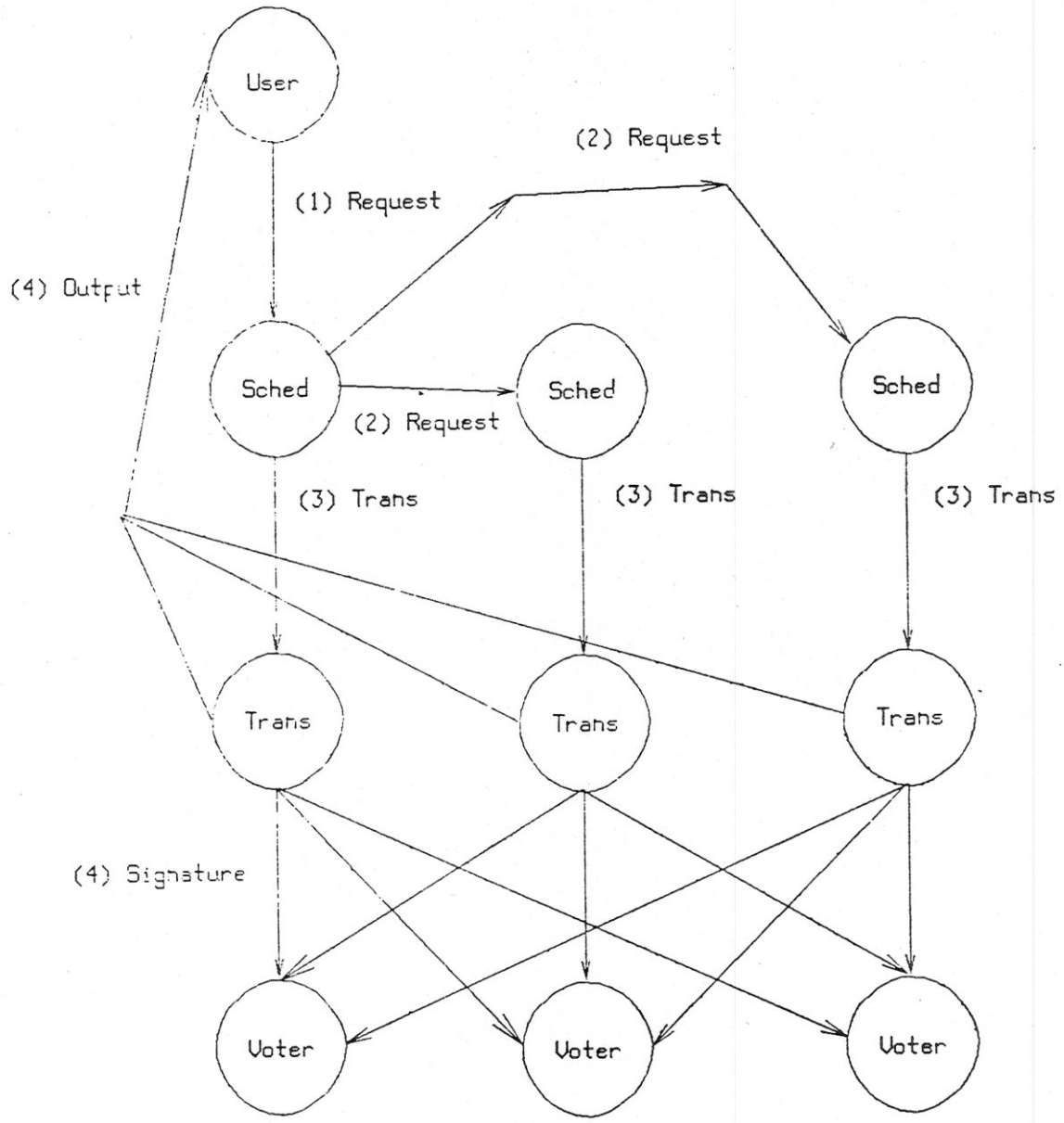

Figure 1. Normal Transaction Processing

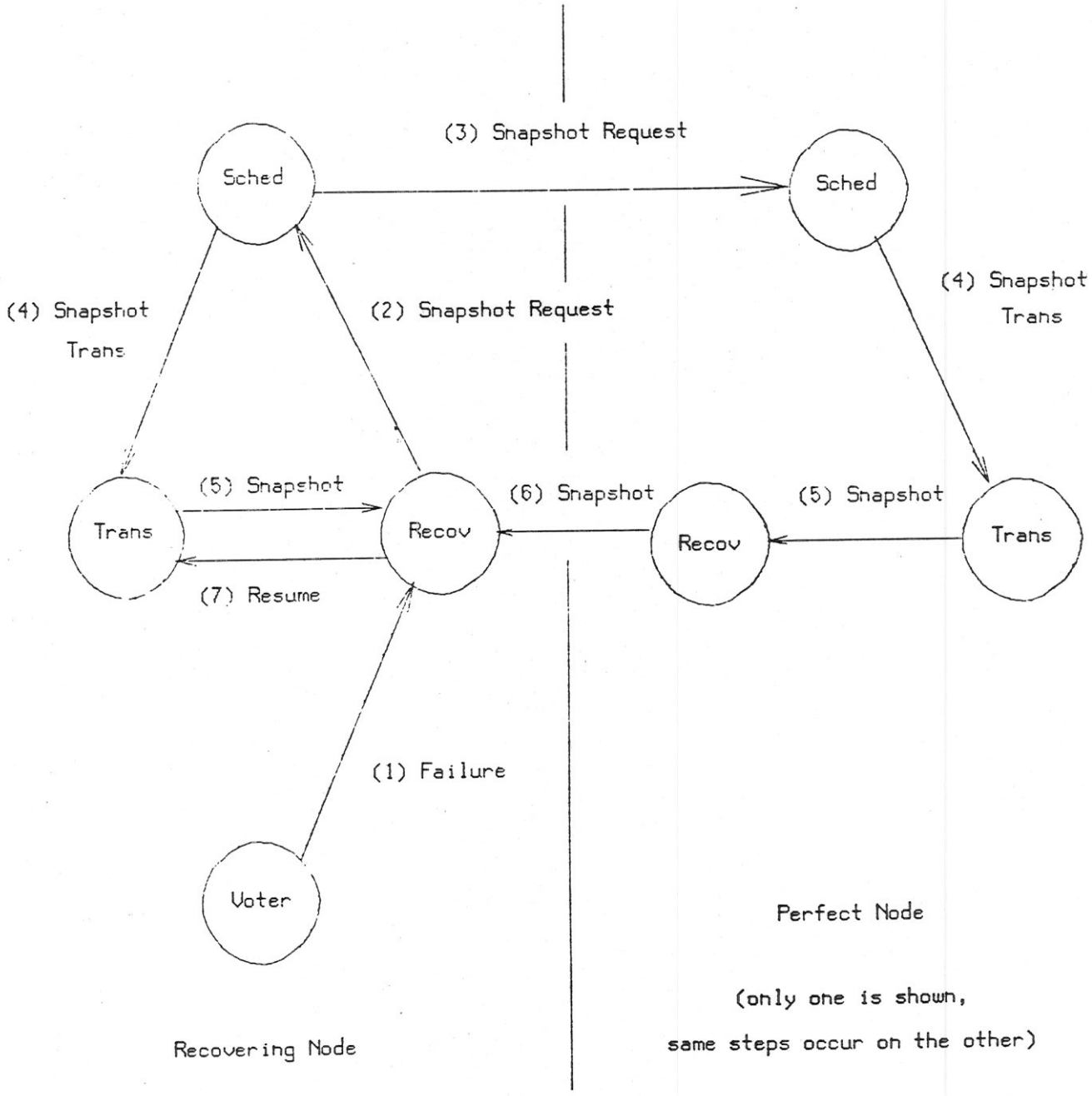

Figure 2. Recovery Processing

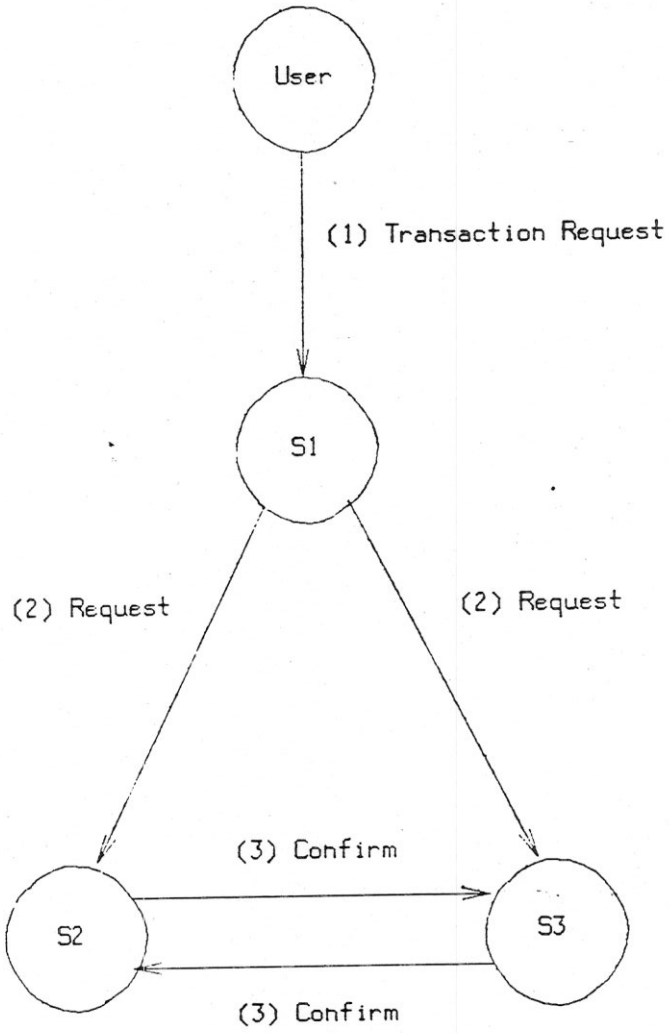

Figure 3. Reliable Broadcast

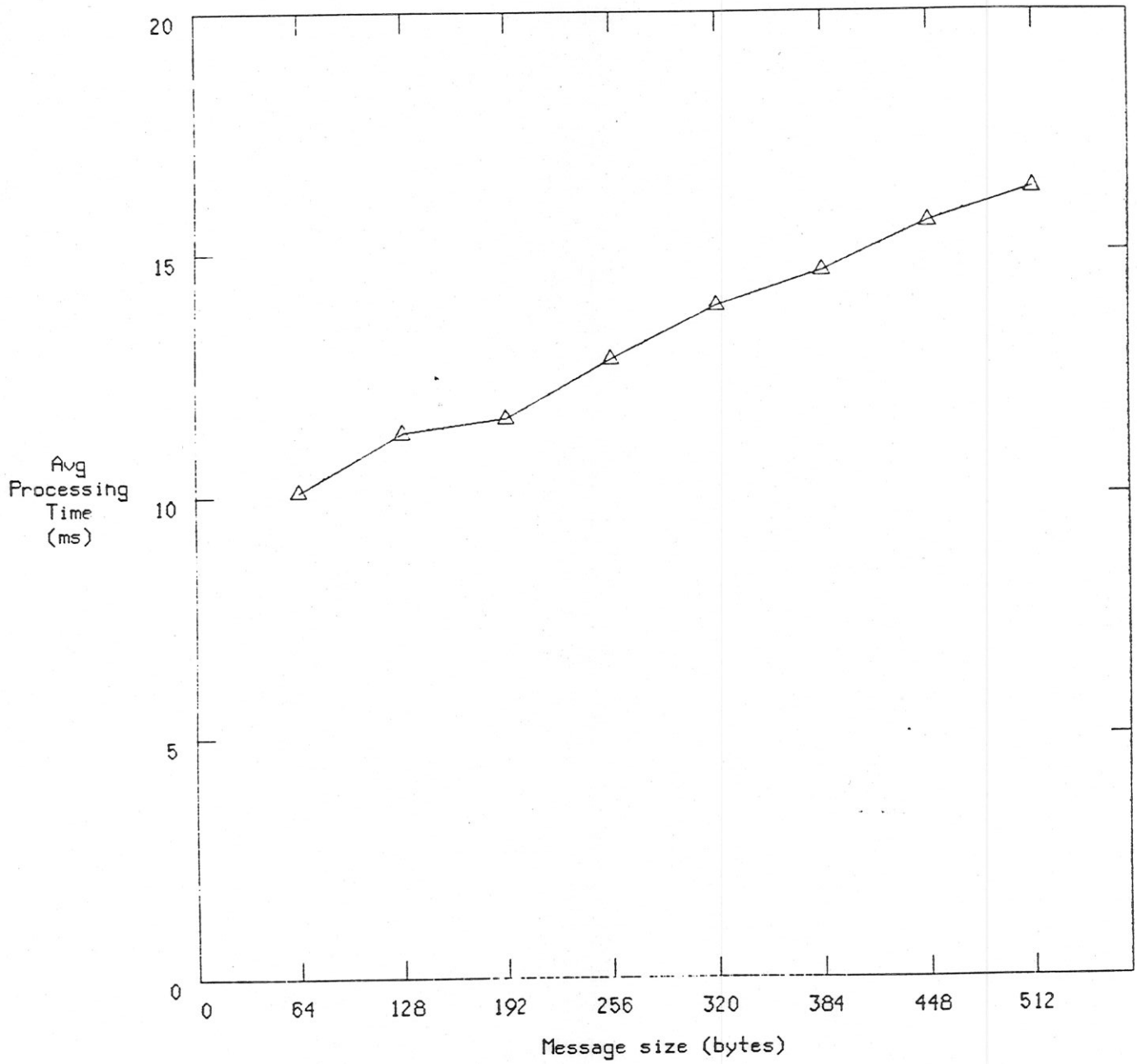

Figure 4. Message Processing Time

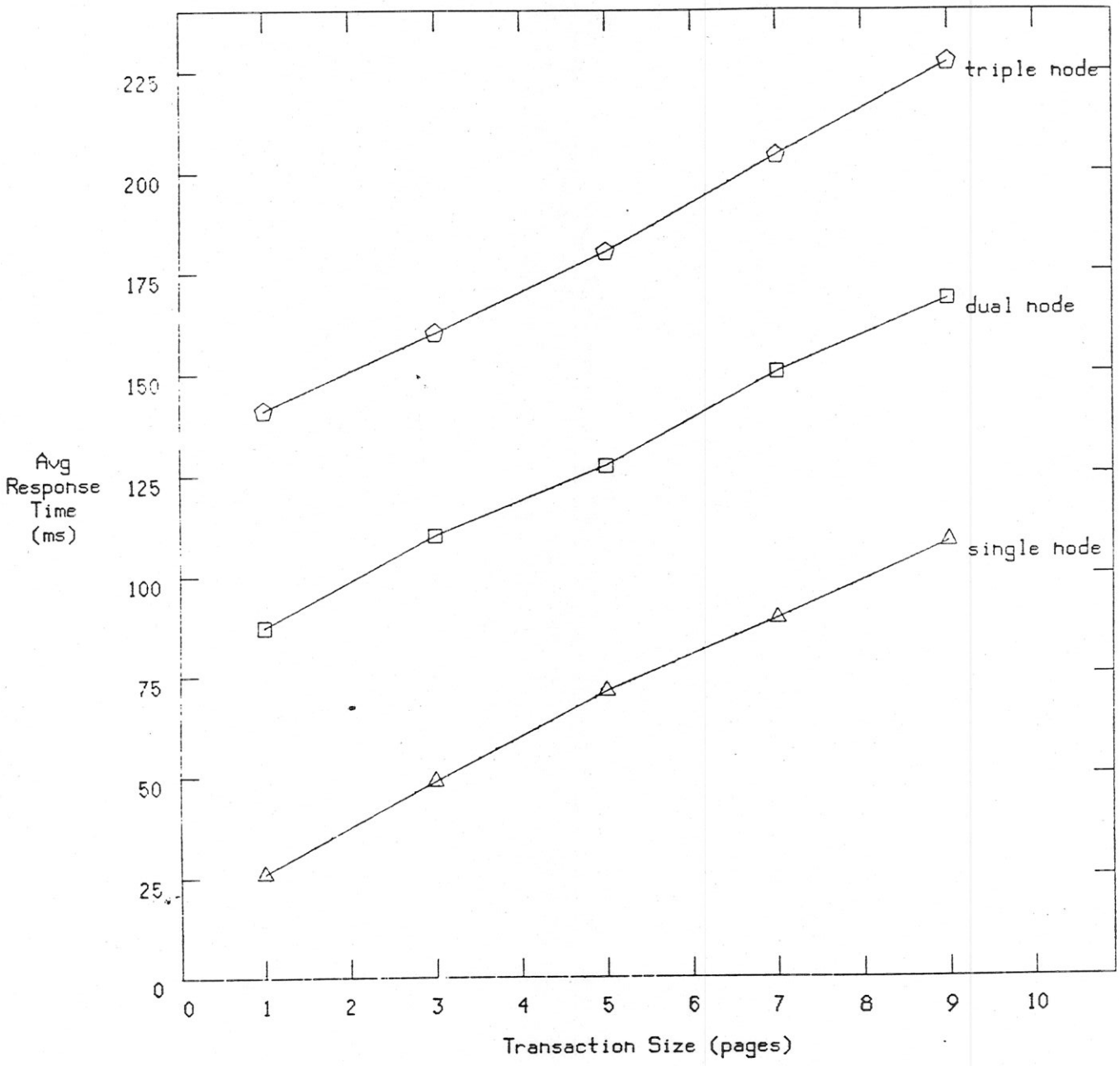

Figure 5. System Response Times (Normal Processing)

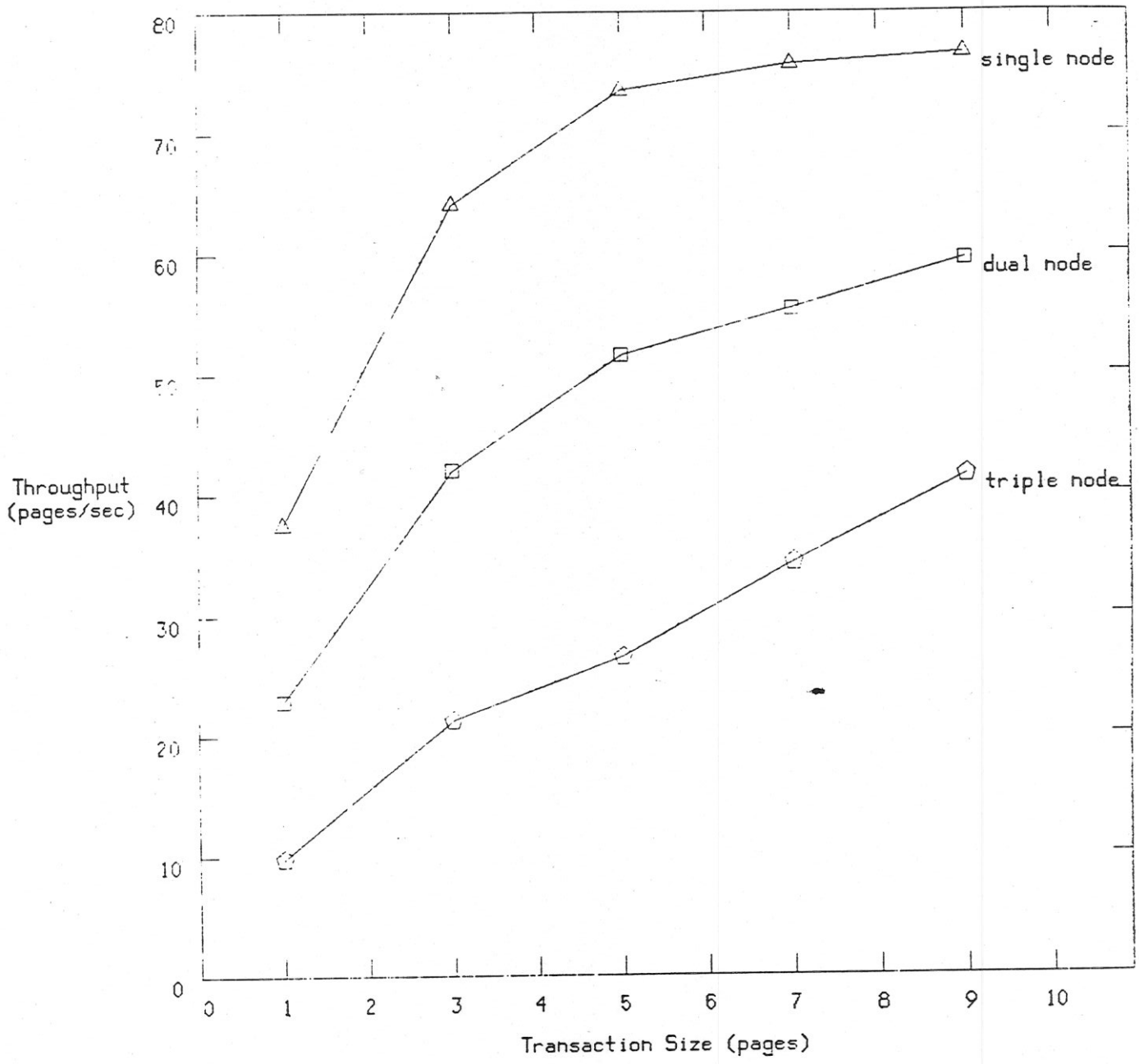

Figure 6. System Throughput (Normal Processing)